Software Designed for AMIGA

Lattice Text Utilities

Lattice
Text Management Utilities (TMU)™
for the
Commodore Amiga

A collection of utilities that provide a language-independent set of tools for examining and editing files.

Lattice, Incorporated
P.O. Box 3072
22W600 Butterfield Road
Glen Ellyn, IL 60137
TWX 910-291-2190
TELEX 532253
FAX (312) 858-8473

COPYRIGHT NOTICE

This software package and document are copyrighted ©1985, 1986, 1987 by Lattice, Incorporated. All rights reserved worldwide. No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language by any means without express written permission.

SINGLE COMPUTER LICENSE

The price paid for one copy of the **Text Management Utilities** licenses you to install and use the product on only one computer at a time. You may remove it from that one computer and install it on another, but at no time are you allowed to make multiple copies available for others to use. You may install **Text Management Utilities** on file servers in a network, but a separate copy must be purchased for each user. See the diskette envelope for additional licensing terms.

DISCLAIMER

Lattice makes no warranties as to the contents of this software product and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Lattice further reserves the right to make changes to the specifications of the program and contents of the manual without obligation to notify any person or organization of such changes.

GETTING STARTED

System Requirements

The Text Management Utilities require a Commodore Amiga personal computer running AmigaDOS version 1.Ø, with at least 256K of memory. Since at the time of this writing only floppy systems are in production, it is assumed that this is your environment.

What's in the Box

When you opened the **TMU** package, you should have found:

- one 3 1/2" floppy disk containing the executables for the utilities: **Grep, Ed, Diff, Wc, Build, Extract, Splat** and **Files;** the **Grep** library (grep.lib); a demonstration of the **Grep** library, (grepdemo.c) with supporting files; grepdemo (pat.h); an installation program (loadtmu); a packing list file (pack.lst); and a read me file (read.me). Take a moment to look at the read.me file to see if there are any changes or additions to this document that you should be aware of.

- this manual with the Lattice registration card. If you did not receive the registration card, let us know at once! This is the only way we can determine whether you are an authorized user, who is eligible to receive free updates. Please complete and return the card.

Setting Up Your Working Disks

Before installing **TMU** onto your system disk, make a backup copy and use that. The original distribution disk should be kept in a safe place, in case an update necessitates its return to Lattice. For backup purposes, you can make additional copies of the distribution disk.

One recommended setup of **TMU** is handled by the **loadtmu** batch file on your distribution disk. To use this batch file, boot up the Amiga, placing the Workbench disk in drive **df0:**. Place the **TMU** distribution disk in drive **df1:**, and invoke:

```
execute df1:loadtmu
```

from the root directory of **df0:**. This batch file will place the executables for **TMU** into your command directory on drive **df0:**.

Using the TMU Program

The demo file **grepdemo**, with its source, **grepdemo.c**, has been provided for the purposes of understanding the functions in the **Grep** library, **grep.lib**. Instructions on how to compile and link the demo are embedded in the comment header of **grepdemo.c**. To run the demo, simply type:

grepdemo

from the directory where this program resides.

The comments on how to create **grepdemo** make the assumption that you are using the Lattice C development system. The file startup sequence included on the distribution disk can be copied into the system directory on your Workbench disk. It makes the logical name assignments which are required when using Lattice C. If you already have a startup sequence customized to your purposes, you can copy the commands from the one on disk into your file with a text editor.

INTRODUCTION

TMU is a collection of utilities, most of which owe their ancestry to the UNIX environment. Together, they provide a language-independent set of tools for examining and editing files. Specifically, the following utilities are provided:

Grep - a pattern matching tool
Ed - a powerful line editor
Diff - a file comparator
Wc - a word count utility
Build - build a batch file from a list of filenames
Extract - extract file names from a directory
Splat - search and replace on multiple files
Files - find files on a disk by their attributes

There is a high degree of cohesion between several of these utilities. **Extract** and **Build** work together in creating a batch file of commands for building software. **Diff** can be made to create a file of differences acceptable to **Ed**, which can work in a batch mode from the created command file.

There are several modes of operation for some of the utilities. As mentioned above, **Ed** can work in either an interactive mode, or can take its commands from a batch file. **Splat** has an option allowing you to see what changes would be made, which proves useful before making changes to your actual source files.

All of the text management utilities are integrated into the Amiga environment. That is, the file specifications given from the command line can use most AmigaDOS wildcards. In addition, MS-DOS wild cards have been provided as an option. For more specifics, see the **read.me** file on the distribution disk.

LATTICE TEXT MAGNAGEMENT UTILITIES

USER'S MANUAL

(FOR THE COMMODORE AMIGA)

TABLE OF CONTENTS

SECTION 1	USING GREP	1-1
1.1	Introduction to **Grep**	1-1
1.2	**Grep** Examples	1-1
1.3	Special Characters	1-2
1.4	Simple Patterns	1-3
1.5	The Wildcard Character	1-4
1.6	Character Classes	1-5
1.7	Closures	1-6
1.8	Anchored Searches	1-8
1.9	Advanced Examples	1-9
1.10	Using the **Grep** Functions in a Program	1-10
1.11	Manual Pages	1-12
1.12	Error Messages	1-20
SECTION 2	USING DIFF AND WC	2-1
2.1	Introduction to **Diff** and **Wc**	2-1
2.2	Options	2-1
2.3	Output Format	2-3
2.4	Implementation Issues	2-5
2.5	Error Messages	2-6
SECTION 3	USING ED	3-1
3.1	Introduction to **Ed**	3-1
3.2	Interrupting **Ed**	3-2
3.3	**Ed** Commands -- Examples	3-3
3.4	Special Characters	3-9
3.5	Patterns and Searching	3-12
3.6	Substitution	3-15
3.7	The **global** Command	3-17
3.8	Batch Mode	3-20
3.9	Error Messages	3-22
3.10	Size Limitations	3-23
3.11	Command Summary	3-24
SECTION 4	USING SPLAT	4-1
4.1	Introduction to **Splat**	4-1
4.2	Using **Splat**	4-1
4.3	Examples	4-2

SECTION 5 USING EXTRACT AND BUILD 5-1

5.1 Introduction to **Extract** and **Build** 5-1
5.2 Using **Extract** 5-1
5.3 Using **Build** 5-2
5.4 Examples 5-2

SECTION 6 USING THE FILES COMMAND 6-1

6.1 Introduction to the **Files Command** 6-1
6.2 Using the **Files Command** 6-1
6.3 Options 6-2

SECTION 7 A NOTE ON WORDSTAR FILES 7-1

INDEX I-1

TRADEMARK ACKNOWLEDGMENTS

Amiga and AmigaDOS are trademarks of Commodore-Amiga, Inc..

Lattice is a registered trademark of Lattice, Inc..

MS-DOS is a registered trademark of Microsoft Corporation.

UNIX is a trademark of AT&T Bell Laboratories.

SECTION 1:
Using Grep

1.1 Introduction to Grep

Grep (global regular expression search and print) is a utility program, familiar to UNIX users, that searches a set of files for a specified pattern and prints each line that contains an expression matching the pattern. **Grep** is a powerful tool that is of great help to software developers, technical writers and computer hobbyists because it is simple to use. It can save a great deal of time and minimize the frustration that often results from attempting to find a word or sequence of words that may occur in a number of files. It's companion programs **Ed** (the Lattice Line Editor) and **Splat** make use of **Grep**'s pattern matching abilities to allow you to develop editing scripts and make complex changes (additions, deletions and substitutions) to a set of files in an automated way.

We shall begin our discussion of **Grep** with some simple examples and proceed to examine the full power of **Grep.**

1.2 Grep Examples

Let's assume that you are a writer who has been editing a file, **article.txt**, and that the file has become quite long. It suddenly occurs to you that, in certain contexts where you should have written "user's guide", you actually typed "users guide". You would like to determine how many occurrences of this mistake there are and roughly where they occur in your file so that you can fix them. As an initial approach you might use the following command:

```
grep guide article.txt
```

Grep will respond by printing all of the lines in the file that have the word "guide" in them. Of course this isn't really what you want and there will most likely be a number of lines that do not contain the expression "users guide" at all but contain such expressions as "this guide", "user's guide", "our guide", "will guide us", and so on. You might instead try:

```
grep users article.txt
```

but this approach suffers from the same drawback since any line containing the word "users" will be printed. What you really want is the following command:

```
grep "users guide" article.txt
```

The entire phrase must be enclosed in quotation marks (" ") so

that **Grep** knows that you want to search for the phrase "users guide". If you use:

```
grep users guide article.txt
```

Grep interprets this as a request to search for the word "users" in the two files **guide** and **article.txt**. The general form of a **Grep** command is:

```
grep [flags] pattern file_1 file_2 ... file_n
```

where **pattern** is the pattern you wish to search for and **file_1**, **file_2** and **file_n** are the files you wish to search. The **flags** parameter is discussed in Section 1.11, let's ignore it for now except to note that the enclosing square brackets ([ ]) mean that the flags parameters are optional.

The patterns that **Grep** can search for are quite general and are technically known as **regular expressions.** However, most people would find the phrase, **regular expression** to be a bit confusing, so the manual will refer to **patterns.** A full and precise description of patterns will be given in Section 1.11. The following sections will be devoted to a more intuitive description of the different sorts of patterns together with a number of detailed examples.

1.3 Special Characters

Before describing the different kinds of patterns, you must be warned that there are a number of characters that **Grep** interprets in a special way. For example, if you were to use the command:

```
grep [ article.txt
```

you would get an error message from **Grep.** This is because the left bracket ([) has a special meaning to **Grep.** This section will mention what the special characters are so that you will be aware of them. Later sections will describe how **Grep** interprets each of these special characters. This section will also show you how to tell **Grep** that you do not want the character interpreted with their special meanings.

Each of the following characters is a special one to **Grep:**

```
!   $   ^   *   -   +   [   ]   .   \
```

The last of these, the backslash (\) is called the **escape character** and is used to escape any special character (including itself) and thereby remove the special sense of that character. Using the example from above, if you wanted to search for the left bracket you would use the command:

```
grep \[ article.txt
```

or if you wanted to search for the phrase "and so on ..." you would use the following command:

```
grep "and so on \.\.\." article.txt
```

To search for the backslash (\) itself use the following:

```
grep \\ article.txt
```

Some of the characters listed above are special only in a certain context, and otherwise function as normal characters. The contexts in which they are special are discussed in detail in later sections, but a good rule of thumb is:

if a Grep command does not behave as you expect it to, try escaping any special characters in the command with the backslash.

In addition to these special characters, there are several sequences of characters (escape sequences) that **Grep** treats in a special way. Each of these begins with a backslash followed by a single letter and their meanings are as follows:

ESCAPE SEQUENCE	MEANING
\n	Newline Character
\s	Space Character
\b	Backspace Character
\t	Tab Character
\\	Escape Character
\xij	Hex Number whose digits are i and j

For example, to locate all lines containing tab characters you would use the command:

```
grep \t article.txt
```

The last escape sequence is provided for those circumstances in which the user might wish to search for the occurrence of a character by using its hex number. For example, to search for occurrences of a control G character, the command would be:

```
grep \x07 article.txt
```

1.4 Simple Patterns

The simplest patterns are the sort we have just seen examples of in the previous section: words, strings of letters and phrases (possibly containing spaces or tab characters). To search for a single word it is sufficient to use just the word as the pattern.

However, to search for a string of words that are separated by spaces you must enclose the entire string within double quotes as in the following command:

```
grep "this is the string to search for" article.txt
```

1.5 The Wildcard Character

The special character, period (.) is treated as a wildcard character by **Grep** in that it is taken to be matched by **any** single character. Thus the pattern:

```
.he
```

will be matched by any of the following:

```
ahe
bhe
che
she
the
```

Note that the dot will be matched by **any** single character, not just by any single alphabetic or numeric character. Thus it will be matched by a space, tab, newline or control character. The wildcard character is one of those you must be careful to keep in mind when you are constructing patterns.

Suppose, for example, that you wanted to see which lines in your file contained the expression "123.45". The command you must use is:

```
grep 123\.45 article.txt
```

where the period (.) is escaped. The pattern:

```
123.45
```

would have matched any of the following expressions:

```
123a45
123 45
123!45
123(45
```

1.6 Character Classes

A **character class** is a pattern that is matched by any character in a given class (or set) of characters. **Grep** uses the left and right bracket symbols ([ ]) to mark the beginning and end of a character class pattern. For example, the expression:

[abc]

denotes a character class that is matched by any of the first three (lower case) letters of the alphabet and the expression:

[Tt]

denotes a character class that is matched by either an upper case T or a lower case t.

Character classes are handy when you want to construct a certain kind of disjunctive pattern. Suppose, for example, that you want to see what lines in your document contain either the word "the" or the word "The". The command:

grep the article.txt

will print out only the lines containing the word "the", while the command:

grep The article.txt

will print out only lines containing "The". But the command:

grep [Tt]he article.txt

will print both sorts of lines. Similarly, the command:

grep [Tt][Hh][Ee] article.txt

will print any line, in which any of the following occur:

THE
THe
The
the
thE
tHE
tHe
ThE

Within a character class, certain symbols are interpreted in special ways. You may use a minus sign (-) to indicate a range of characters, as in:

[a-z]

which is matched by any lower case letter in the English alphabet or:

[a-zA-Z0-9]

which is matched by any lower case letter, any upper case letter or any decimal numeral.

A simple, yet practical, use of character classes arises when you want to see all the lines in your file where you refer to a specific year. The command:

grep [0-9][0-9][0-9][0-9] article.txt

would accomplish this. If you are concerned only with years in the 20th century then use:

grep 19[0-9][0-9] article.txt

If you are concerned with years in the 19th and 20th centuries then use the following command:

grep 1[89][0-9][0-9] article.txt

The exclamation point (**!**) is treated as special when it occurs as the first character in a character class. In this case it acts as a **negation operator**, so the pattern:

[!a-z]

will be matched by any character that is **not** a lower case letter of the English alphabet, and the pattern:

[!#]

will be matched by any character that is **not** a number sign. If an exclamation point (**!**) occurs in a character class elsewhere than in the first position it does not have this special sense.

As you will see in Section 1.9, character classes lend a great deal of power to **Grep** and allow us to search for some very complex patterns.

1.7 Closures

The symbols, star (*****) and plus sign (**+**) are used to denote what are called **closures.** The star (*****) may be thought of as meaning **zero or more of.** The plus sign (**+**) may be thought of as meaning **one or more of.**

Suppose that in your file you have referred to Columbus and mentioned a year in which some event has occurred. You don't want to see every line in which the name Columbus occurs, nor do you want to see every line in which a yearly date occurs. You want to see only those lines in which both Columbus and a yearly date occur. The command would then be:

```
grep "Columbus.*[Ø-9][Ø-9][Ø-9][Ø-9]" article.txt
```

Within the pattern, the expression **.*** would be matched by any string containing zero or more occurrences of any character. The entire pattern will therefore match such expressions as:

```
Columbus-5555
Columbus sets sail in 1492
Columbus, Ohio was founded before 198Ø
```

and so on. When there is a choice to be made concerning how long the pattern is that matches a closure, **Grep** chooses to match the longest possible pattern.

To consider another simple example, suppose that you wanted to see all the lines containing any of the following words:

```
the
The
she
She
he
```

If you use the pattern:

```
[tTsS]*he
```

it will be matched by the following examples:

```
ttttthe
TsShe
Sthe
TTTSSStthe
```

It would be nice to detect these obvious typographical errors in any event. However, notice that this pattern would be matched by **any** expression containing "he" because of the zero or more qualifier on the character class. A better pattern would be the following:

```
[!a-zA-Z][tTsS]*he[!a-zA-Z]
```

To eliminate those contexts when "he" is embedded in a longer word, or to ensure that the word is isolated by spaces use the following:

```
" [tTsS]*he "
```

The plus sign symbol (+) works precisely the same as the star (*) except for its requirement that at least one occurrence of its character or character class be present. Thus:

```
[tTsS]*he
```

will be matched by "he" but the following will not:

```
[tTsS]+he
```

1.8 Anchored Searches

Frequently you may wish to see only those lines in a file in which a given pattern occurs at the beginning of the line. The special character carat (^), when it is the first character in a pattern, indicates that the pattern will be matched only if it begins a line in the file. Thus, to find all of those lines in your file that begin with a space or a tab character, you should use the command:

```
grep "^[ \t]" article.txt
```

Note that the quotation marks (" ") are necessary in this pattern since a space occurs within the pattern itself. Similarly, to find all lines that begin with numerals (a section heading, for example) use the following:

```
grep ^[0-9] article.txt
```

The special character, dollar sign ($) works much like the carat (^) except it forces the match to occur only at the END of a line. If, for example, you wanted to search for all occurences of the C language comment terminator */ at the end of a line you would use the following:

```
grep \*/$ article.txt
```

Here the backslash (\) is used to escape the star (*) which otherwise would be interpreted in its special sense as a closure operator.

1.9 Advanced Examples

This section will provide some additional examples of using **Grep** to search files for patterns. It will concentrate on searching for files containing source code text, in particular, C language source code.

Example 1

Suppose you have a number of C language source files, all of whose file names have a .c extension, and you need to search these files to find all calls of either of the functions "read" or "fread". Use the following command:

```
grep "[f]*read[ ]*(" #?.c
```

This will cause a printing of each line, in any of the files, that contain either read or fread, followed by zero or more spaces, followed by a left parenthesis. This should display all of the lines you are interested in.

Example 2

To illustrate a second example, suppose you want to search your files for every function definition. This really is not possible unless you make some reasonable assumptions. However, it is common for most C programmers to define a function by first placing the name and parameter list of the function on a separate line, then following this with the body of the function. Making this assumption, use the command:

```
grep "^[a-zA-Z0-9_]+[ ]*(" #?.c
```

The pattern above will match only an expression that begins a line. Then the expression must consist of one or more lower case letters, upper case letters or the underscore. This must be followed by zero or more spaces and then a left parenthesis. This pattern will pick out lines like the ones shown:

```
getl(s)         /* get a line */
_read ()
my_read(fp) { fread(fp,x);}
```

however, it will not select a line such as:

```
char *getl(p)    /* get a line */
```

which may be the beginning of a function definition.

Example 3

Let's say you are having trouble compiling a C program because the compiler complains that there is a mismatch between left and right braces ({ }). The program is very long and you have been unable to determine where things have gone wrong. You decide to use **Grep** to help by printing each line that contains either a left or right brace. This way the error may be easier to spot. The command to use is:

```
grep [{}] #?.c
```

Example 4

If you want to find all occurrences of strings in your files -- i.e., sequences of characters enclosed in double quotation marks (" "). The appropriate command is:

```
grep \".*\" #?.c
```

1.10 Using the Grep Functions in a Program

The **Grep** utility makes use of several functions that construct an internal representation of a pattern from a string of characters and then checks to see if that pattern is matched in a given string. You are supplied with files on the distribution disk that allow you to make use of these functions in C programs. The functions are included in the library file, **grep.lib** (assuming that you are using the Lattice C Compiler). This allows you to build the same sophisiticated pattern matching capability into your own programs as is present in **Grep**. This section will review the procedures to accomplish this goal.

The files in the **Grep** library are: **re_gen.o**, **re_match.o** and **re_smatc.o**. These contain the functions **re_gen()**, **re_match()**, **are_match()**, **re_smatch()**, and **are_smatch()**.

The use of these functions will become clear with a simple example. Suppose you have a global array of strings (i.e., of pointers to characters), and need to have a function that will search this array for strings that contain a match for a given pattern. In addition, let's assume that the pattern you wish to search for may be described as "any sequence of characters (including the empty sequence) enclosed in parentheses". Finally, let's assume that the size of the array is defined by the constant "ARYMAX". Your code should then look something like:

```
extern char *ary_str[];   /* array of string pointers
                           *  assumed to be defined and
                           *  filled elsewhere
                           */

/*
 * Define a function findpars() that will search ary_str for
 * members containing a string of characters enclosed in
 * parentheses.  For each such member of ary_str, print
 * its array index.
 *
 */

findpars()
{
     int i;

     for ( i = 0; i < ARYMAX; i++ )
          if ( re_smatch("(.*)",ary_str[i]) >= 0 )
               printf("Found match at index %d\n",i);
}
```

This will work, but it is inefficient. Every time **re_smatch()** is called, it calls **re_gen()** to generate an internal representation of a pattern. In the example the pattern never changes, so why regenerate it each time around the loop? A better definition of **findpars()** would be:

```
#include "pat.h"    /* Need to include this header to have
                     * 'PATTERN' defined
                     */
findpars()
{
     PATTERN re_gen();    /*  must declare re_gen() as
                           *  returning a PATTERN
                           */
     PATTERN p;      /* to point to the pattern generated */
     int i;

     if ( (p = re_gen("(.*)")) == NULL ) {
          printf("Error in generating pattern\n");
          return;
     }

     /* pattern was generated okay */

     for ( i = 0; i < ARYMAX; i++ )
          if ( re_match(ary_str[i],p) >= 0 )
               printf("Found a match at index %d\n",i);
}
```

This example should suffice to show how the supplied functions may be use in programs where the pattern-matching power of **Grep** is required. See Section 1.11 for more details on these functions. Of course, you must be sure to link the appropriate object modules in when you link your program together. If you are using the Lattice C Compiler with the Metacomco linker, then the command line to the linker should appear:

```
LINK:alink FROM LIB:c.o+myprog.o TO myprog
           LIBRARY grep.lib+LIB:lc.lib+LIB:amiga.lib
```

assuming that **myprog.o** is the object module you have compiled which invokes calls to functions in the **Grep** library. The command line above assumes that you are using the startup-sequence that is provided with the Lattice C compiler, which makes some logical name assignments to directories.

1.11 Manual Pages

This section contains UNIX-style manual pages for **Grep** and its associated functions that may serve both as a quick reference and as a precise and exhaustive definition of the sort of patterns that **Grep** deals with. In addition, the manual page on **Grep** itself describes a number of flags that may be used when employing the **Grep** command.

NAME

grep -- global regular expression search and print

SYNOPSIS

grep [>outfile] [-p] [-n] [-c] [-f]
[-v] [-s] [-V] [-$] [-m] pat filel ... filen

DESCRIPTION

This version of **Grep** is very much like the UNIX grep, but with a few exceptions. A slightly less general class of patterns (regular expressions) is supported. In particular, the UNIX regular expressions of the form:

\{m\} or **\{m,\}** or **\{m,n\}**

which match ranges of occurrences of regular expressions are not recognized by this **Grep.** Neither are the regular expressions of the forms:

\(... \) or **\n**

where n is a numeral, and **\n** is intended to match the same string of characters as was matched by the expression enclosed between **\(** and **\)**.

Aside from these departures, this **Grep** has all the features of the UNIX grep. In particular:

(1) An ordinary character is a one-character pattern that matches itself.

(2) A backslash (\) followed by any special character is a one-character pattern that matches the special character itself, EXCEPT when it is used in an escape sequence for a non-graphic character as described in (6) below.

The special characters are:

(a) **.**, *****, **[**, and ****. Unlike UNIX, **.**, *****, and **** remain special even within square brackets, but **[** does not.

(b) **^**, which is special at the BEGINNING of an ENTIRE pattern.

(c) **$**, which is special at the END of an entire pattern (see below).

(d) **!**, which is special immediately following a [(see below).

(3) A period (.) is a one-character pattern that matches any character except newline or end-of-string.

(4) A non-empty string of characters enclosed in square brackets ([ ]) is a one-character pattern that matches ANY ONE character in that string. However, if the first character of the string is **!** then the one-character pattern matches any character EXCEPT newline or end-of-string, and the remaining characters in the string. Thus **!** is a "not" sign applied to a character class. Note that this is a notational departure from UNIX, which uses ^ for this application (thus rendering ^ ambiguous).

Also supported are such character classes as:

[a-z]

indicating all of the letters 'a' through 'z'. The only restriction in such cases is that the first character must occur alphabetically prior to the second.

(5) **Grep** recognizes certain non-graphic characters denoted in the manner of Kernighan & Ritchie. In particular, the expression on the left below may be used to denote a pattern that is matched by the character described on the right:

\n	newline
\t	tab
\b	backspace
\r	carriage return
****	backslash
\xmn	character whose hex number is denoted by the hex numerals m and n (in sequence).

(6) Patterns containing whitespace (spaces and tabs) may be specified to **Grep** by enclosing them in double quote marks (" "). Thus a string may be grepped for by:

grep "this is a string" foo.c

Because of the way in which the command line given to AmigaDOS is available to **Grep**, any tab in the command line will remain a tab when handed to **Grep** (it will not be converted to spaces). Also, spaces, as well as any combination of tabs and spaces, will be handed to **Grep** in exactly the manner they are entered from the command line.

It is also allowable to use the escaped form of the tab character **\t**, inside a quoted string handed to **Grep**. If it is necessary to **Grep** for an expression containing the double quote itself, this character may be escaped with the backslash as shown in the following example:

```
grep "this : \" is a double quote" foo.c
```

(7) If r is a pattern, then:

r*

matches ZERO OR MORE occurrences of **r**. If there is a choice, then the longest leftmost string that matches **r** is chosen.

(8) If r is a pattern, then:

r+

matches ONE OR MORE occurrences of **r**. If there is a choice, then the longest leftmost string that matches **r** is chosen.

(9) The concatenation of patterns is a pattern that matches the concatenation of the strings matched by each component of the pattern.

Grep also supports several flags on the command line. These are:

-V print version number of **Grep**.

-m use MS-DOS file name patterns (the default is to use AmigaDOS file name patterns and wildcards).

-n turn off the numbering of matched lines (the default, opposite to UNIX, is that each matched line is preceded by its line number when displayed).

-f print only the names of the files in which a match of the pattern was found.

-c print only a count of the matched lines.

-v print only the lines in which a match of the pattern is NOT found (can be used with **-n**, **-f** and **-c**).

-s "show all file names". Normally **Grep** will display only those file names where it has found a match for the expression. This option forces the printing of even those file names in which no match was found.

-p filter out non-printable characters. This option is useful when **grep**ping WordStar files or other files in which control characters may occur. Only the normal printable ASCII characters will be displayed.

-$ ignore case distinctions. Normally **Grep** distinguishes between upper and lower case. Use of this flag forces **Grep** to ignore such a distinction. (For example, each of "int", "INT", "Int", "INt", etc. would match the pattern "int" if the **-$** option were used.)

Output from **Grep** may be redirected to a file by using:

>filename

as the first command-line argument to **Grep**, where "filename" stands for the name of the desired output file.

Files to be searched may be specified to **Grep** in any manner understood by AmigaDOS. The AmigaDOS wildcard patterns, which are enumerated in the AmigaDOS User's Manual under the LIST command, can be used to specify groups of files to be **grep**ped. For example:

df0:#?.c will match all files on drive df0:, with file extension **c**.

g#?.c will match all files in the current directory with extension **c**, beginning with **g**.

Any of the file specifications to **Grep** can also be combined, such as:

#?.c s/#?.c will match all files both in the current directory and in the subdirectory **s/**, with file extension **c**.

If you are more comfortable with MS-DOS file naming conventions and wildcards, say from a previous implementation of **Grep**, you can use these conventions by toggling the **-m** switch from the command line. An example of this usage of **Grep** would be:

grep -m pat *.c will match all files in the current directory with file extension **c**.

The characters recognized as wildcards with the **-m** usage are:

***** matches 0 or more of any character.

? matches any character.

NAME

re_gen -- generate a regular expression pattern

SYNOPSIS

```
pat = re_gen(str)
PATTERN pat;     pattern generated
char *str;       character string representation of
                 the pattern
```

DESCRIPTION

Translates the string representation (str) of a pattern into an internal representation (pat) described in pat.h. This PATTERN may then be used by the functions re_match() and are_match() to match against a string.

RETURNS

the PATTERN, if successful; otherwise NULL

SEE ALSO

grep, re_match()

NAME

re_match -- regular expression match
are_match -- anchored regular expression match

SYNOPSIS

```
found = re_match(str,pat)
found = are_match(str,index,pat)
int found;
int index;              index to anchor match
PATTERN pat;            regular expression pattern
char *str;              string being scanned
```

DESCRIPTION

Scans the specified string to determine if it contains an occurrence of the given pattern. The pattern is an internal representation of a regular expression (RE), usually obtained via a call to the function re_gen(). Note that you must #include the file pat.h in order to declare any variable of type PATTERN.

RETURNS

found = the index of the last character in the first substring of str that matches pat, if there is one
= -1 if there is no match

SEE ALSO

re_gen(), re_smatch()

NAME

re_smatch -- regular expression match of strings
are_smatch -- anchored regular expression match of strings

SYNOPSIS

```
found = re_smatch(str1,str2)
found = are_smatch(index,str1,str2)
int found;
int index;          index to anchor match
char *str1;         pattern string
char *str2;         string being scanned
```

DESCRIPTION

Scans the specified string (str2) to determine if it contains an occurrence of the given pattern (str1). The string (str2) denotes a pattern as characterized in previous sections. are_smatch() is like re_smatch() except that it performs an anchored search from index. These functions are similar to re_match() and are_match() with the exception that the pattern specified in re_match() and are_match() is the internal representation of a pattern produced by re_gen().

RETURNS

found = the index of the last character in the first substring of str2 that matches str1, if there is one
= -1 if there is no match

SEE ALSO

re_match()

1.12 Error Messages

Bad character class

Grep is unable to interpret a character class in the pattern you have asked it to find. Check to see if you have used any special symbols that you did not intend to. This message can also be generated if in specifying a character class and using the - character to indicate a range of characters, the first character in the range does not alphabetically precede the second (e.g., [d-a]).

Bad pattern

Grep is unable to interpret the pattern you are asking it to find. Most likely this is a result of the occurrence of a special character in the pattern that does not make sense to **Grep** in that context. Check to see if you have forgotten to escape a special character.

Can't find file(s) . . .

Grep is unable to find one or more of the files you have asked it to search. Most likely, they are not there or you have mistyped the file names.

Can't open . . .

Grep is unable to open the named file. This could be because the file is not there at all or that it is protected.

Closing] not found

Grep believes that you have asked it to search for a pattern that contains a character class but that you have left off the right bracket (**]**) that terminates the character class. If this is not so, perhaps you wanted the left bracket ([) to form part of the pattern but forgot to escape it to remove its special sense.

Empty character class

A character class you have used in your pattern has no members. (e.g., [ ]).

Invalid option

You have used a command line option that **Grep** does not recognize. This message can also be generated if your pattern begins with a minus sign (-) but you have not escaped it nor enclosed the pattern in double quote marks (" ").

Improper hex specification

You have used the **\x** escape sequence but have not followed it with two recognizable hex digits.

Incompatible combination of options

The options with which you have invoked **Grep** are asking it to do something contradictory.

No beginning double quote in pattern

Grep has detected a double quote (") in a pattern which was not started with a double quote. Perhaps you meant to escape it.

No file arguments provided

Grep believes that you have invoked it with a pattern but no file names. This can also happen if you have invoked it with a file name but no pattern.

No pattern or file arguments given

Grep can't seem to find either a pattern or file name on its command line.

Out of space

There are several out of space messages you may see. They mean that **Grep** has run out of space in constructing its pattern representation. But, if you see this message send in a sample that causes the problem so we can fix it in the next release.

Pattern ill-formed: no terminating double quote

You started the pattern with a double quote (") but did not terminate it with one.

Too few arguments to Grep

Grep demands at least two arguments on its command line, a pattern and at least one file name.

In addition to the above error messages, there are some internal error messages that you should never see. If you get any error message differing notably from the above, call Lattice to report it, or send us a disk and a letter. You should be prepared to provide the following information:

(1) The version of **Grep** you are using;

(2) The exact wording of the command line you used; and

(3) The exact wording of the message you got.

SECTION 2:
Using Diff and Wc

2.1 Introduction to DIFF and WC

Diff (differential file comparator) is a utility that will determine the differences between two files and then display these differences in an informative way. The proper way of invoking **Diff** is:

diff [options] newfile oldfile

Diff normally responds by displaying its output on the screen, but output may be redirected to a file in the usual AmigaDOS fashion or by using the -o option described below. The simple command:

diff

will result in a usage message being printed, informing you of the options available and the version of **Diff** you are using.

Wc (word count) is a companion utility to **Diff** that will count the number of characters, words and lines in a file. The concept of **word** employed by **Wc** is a rather intuitive one: a word ends when whitespace (a space, tab, carriage return or linefeed) is encountered. From this perspective a word is a sequence of printable characters. A line ends with a newline character. Since **Wc** works in **translated mode**, combinations of a newline followed by a carriage return are collapsed to a single linefeed. The way to use **Wc** is:

wc [-p] [-s] file

and **Wc** will respond by displaying counts of the number of characters, words, and lines in the named file. The options are described in the next section. **Wc** is a quick and handy way of getting some basic statistics on a file, and it can be helpful in using **Diff** with the **-l** option by telling you how large **Diff's** line number tables must be.

2.2 Options

There are several options that may be used with **Diff**. These are:

-ofile The output from **Diff** will be directed into the

named file.

-lnumber This option sets the size of the table that **Diff** uses for the lines in each file to **number.** By default, **Diff** assumes that the files being compared will have a maximum of 1000 lines in each file and creates two tables (one for each file) of size 1000. If the files you are comparing are larger than 1000 lines, you may use this option to increase the size of the tables. If **Diff** seems to be running out of memory in comparing your files and your files are less than 1000 lines in length. You may use this option to decrease the size of the tables and leave more memory available for **Diff** to use.

-p Filter printable characters. This option is identical to the **-p** option of **Grep,** and filters non-printable characters out of the input stream. It is useful for **diff**ing WordStar files.

-w This option causes **Diff** to ignore differences pertaining only to whitespace (spaces or tabs). When two lines are compared using this option, trailing blanks are removed and sequences of whitespace are compressed to a single space.

-e When this option is used **Diff** will generate output suitable for use with the Lattice Line Editor. That is, an **editor script** is generated which may be used by the Lattice Line Editor to convert **oldfile** into **newfile.**

-b**size** Set the size of the I/O buffer to **size.** The default is 4K.

Wc has two options. The **-p** option is identical to that option in **Grep** and **Diff.** Without it, running **Wc** on a WordStar file will likely be a bit inaccurate. When the **-s** option is employed, as in:

wc -s file

then in addition to the information concerning the number of characters, words and lines in the given file, **Wc** will calculate a **checksum** for the file. This checksum is calculated in two parts. The first part is simply the number of printable ASCII characters in the file. The second part is computed with the help of an internal table that matches printable ASCII characters uniquely to numbers (the exact correspondence is unimportant). Each time a printable ASCII character (excluding whitespace) is encountered in the file, its numeric representative from the

table is added to the sum. The final sum is the second part of the checksum reported by **Wc**. Thus the checksum consists of the two parts, and is displayed in the following format:

number_of_printable_ASCII_characters sum

This checksum provided by **Wc** can be useful in one of two circumstances. First, it provides a quick check on whether two files are identical. If their checksums differ, then they are **not** identical. Second, it can be used in the context of uploading or downloading a file from one machine to another to ensure that no inadvertant character translation has taken place. This can be troublesome when, uploading a text file from the Amiga to an IBM mainframe computer that uses the EBCDIC character set, or in downloading from such a mainframe to the Amiga. Since **Wc**'s character representation table is independent of the character set used, and since it ignores whitespace, then the result of running **Wc** on a file on the Amiga and on the uploaded file on the mainframe should be that the checksums are identical. Of course to do this you must have **Wc** on both the host and target machines. For that reason, the source of **Wc** is provided. You need only transfer it to the mainframe and compile it there. It is sufficiently small that you could enter it from a terminal. Many hours of debugging can be avoided by comparing checksums after file transfer.

2.3 Output Format

Diff has two output formats, depending upon whether the **-e** flag is being used. The more common of the two formats, when the **-e** flag is **not** being used, is as follows. After giving the command:

diff newfile oldfile

Diff responds first by displaying:

TO TRANSFORM oldfile INTO newfile ...

followed by a sequence of three different kinds of change blocks. The three types of change blocks are:

(1) A **delete** block of the form:

```
*** DELETE [i,j] FROM oldfile ***
<aaa
<bbb
<ccc
  .
  .
  .
```

where "aaa", "bbb", "ccc", etc. will be the text of

the lines to be deleted from oldfile, and i and j are the beginning and ending line numbers in **oldfile** of the block to be deleted. The less then symbol (<) preceding the line text indicates that a deletion is to take place.

(2) An **append** block of the form:

```
*** APPEND AFTER i IN oldfile ***
>aaa
>bbb
>ccc
  .
  .
  .
```

where the greater than symbol (>) indicates that an appending is to take place, and i is the line in oldfile after which the lines represented by "aaa", "aaa", "bbb", "ccc", etc. are to be appended.

(3) A **change** block of the form:

```
*** CHANGE [i,j] IN oldfile TO [m,n] IN newfile ***
<aaa
<bbb
<ccc
  .
  .
  .
---------------
>xxx
>yyy
>zzz
  .
  .
  .
```

where the block of lines (numbers i through j in oldfile) represented by "aaa", "bbb", "ccc", etc. are to be changed to the block of lines (numbers m through n in newfile) represented by "xxx", "yyy", "zzz", etc. This replacement is equivalent to a deletion of the first block and an appending of the second.

The second output format used by **Diff** is generated when the **-e** flag is used, and this causes the output to be generated in a form that is readable as a command file to the Lattice Line Editor (it is also readable by the UNIX line editor, ed). The format is similar to that described above with the omission of the commentary lines marked by initial and terminal "***" strings, and without the < > markers indicating deletes and

appends. In place of the commentary lines are editor commands such as "5,8d", "23a", or "20,25c". The text of lines to be deleted from **oldfile** does not, of course, appear. In comparing the normally formatted output to that generated with the **-e** option, do not expect the line numbers mentioned in the two formats to correspond very closely. Since the editor would be making changes dynamically in accord with the editing script, the number of any given line in the new version of the file would be subject to change as lines prior to it are deleted or inserted.

2.4 Implementation Issues

There are several different designs available for a file comparator program. The one used here is described in "A Technique for Isolating Diferences Between Files" by Paul Heckel (Communications of the ACM, April 1978, pp. 264-268).

The matching algorithm described by Heckel is simple, fast, and effective. Briefly it works as follows: Identify those lines that occur exactly once in each file. Assume that these lines match one another. Now sweep downward in the first file. Each time you come to a line L that is already matched to one (L') in the second file, examine the very next line (L+1). If it is the same as the corresponding line (L'+1) in the second file, then match those lines as well. Once that downward sweep is completed, sweep upward in the first file attempting to match in the same way (except you look at L-1 and L'-1 when finding a matched line L). You have now completed matching the two files.

There are a couple of problems with this algorithm that a knowledgeable user may wish to keep in mind. Notice that the effect of the matching algorithm is to match blocks of text to one another in the two files. However, as just described, the algorithm frequently will cross match single lines or two blocks of lines. That is, lines L and L+k may be matched to lines L' and L'-j. Consequently, after the process described above is completed, the matches must be "untangled" in order to generate a coherent report of the differences. The principle used is to look at two cross matched blocks and then "unmatch" the smaller of the blocks. This is an attempt to satisfy the intuition that the differences reported should strive to be minimal.

The second problem is that since the algorithm must first seed itself by identifying lines that occur exactly once in each file, it is possible for it to fail to match any lines even when there is an obvious match. Consider the following two files:

aaa	**bbb**
xxx	**xxx**
xxx	**xxx**
yyy	**zzz**

The algorithm will fail to match the blocks of "xxx" lines since it doesn't find any lines that occur exactly once in each file to seed its block matching algorithm. Such an arrangement of lines in normal text files is highly improbable. But should something like this occur, you will be aware of the reason for it.

In addition to these algorithmic points, there are some considerations of memory size that the user may wish to keep in mind. The program uses two tables (one for each file) to keep track of the lines read in from each file. The default size of each of these tables is 1000, but (as mentioned above) this may be either increased or decreased by use of the **-l** option. In addition, if the **-w** (ignore differences in whitespace) option is invoked, more memory is required since the program needs to keep a representative of each line in both its compressed and original form. Thus the **-w** option forces creation of a third table used in the storage and comparison of lines.

Note, in all cases only the minimal number of strings will be stored since only one copy of a line's text is ever stored.

The program also will dynamically allocate storage for any line text it needs to store. And finally, the program uses an input/output buffer that is 8K in size. There are, therefore, several circumstances in which **Diff** may run out of space.

Should **Diff** run out of space, there are several responses that the user may employ. First, you may attempt to decrease the size of the line tables by employing the -l option. Second, if you are using the **-w** option, you may try to use **Diff** without this option. Third, you may decrease the size of the I/O buffer by using the **-b** option. (There may be some speed degradation in this case since more disk accesses may be required.)

For the most part, however, the regular **Diff** performs quite nicely without any of these additional encouragements.

2.5 Error Messages

The following error messages may be generated by **Diff**.

Can't open . . .

Diff is unable to open the named file. This could be because the file is not there at all or because it is protected.

Diff I/O Error

This is a catch-all error message that does not necessarily mean that there has been an I/O problem (though most likely this is the cause of the message).

Improper -l specification: . . .

The specification you have given to the **-l** option cannot be interpreted as a number by **Diff.**

Internal Error: . . .

You should never see a message of this sort. If you do, then it means that something is wrong with the design of **Diff.** Contact Lattice and be prepared to provide the following information:

(1) the version of **Diff** you are using;
(2) the exact wording of the command line you used;
(3) the exact wording of the message you got; and
(4) the size of the files you were trying to compare.

Line table overflow

The line table is too small to hold the number of lines in one of the files being compared. Try increasing the line table size with the **-l** option.

Not enough memory for ... lines per file

You have used the **-l** option to increase the line table size, but you do not have enough memory to increase it by the amount specified.

Out of memory

The meaning of this is fairly obvious. Try decreasing the line table size with the **-l** option if that is possible. Don't use the **-w** option.

Too many file names: . . .

You have made an error on the command line in such a way that **Diff** thinks you are asking it to operate on more than two files.

Unrecognized option

You have attempted to invoke an option that **Diff** does not recognize.

SECTION 3:
Using ED

3.1 Introduction to Ed

Ed is a text editor program that is used to create or to modify a file consisting of text. Text is simply sequences of characters (letters, numbers, etc.) that may be displayed on the screen of a monitor or printed on a printer. This particular text editor is patterned closely upon the UNIX text editor of the same name and supports most of the features found in the UNIX editor. In addition, however, this editor may be used to operate in a batch mode in which a file of editor commands are automatically executed on a series of named files. This is a handy feature in those cases where you wish to make exactly the same changes in a large number of files, since it allows you to make those changes without invoking the editor and going through the same motions for each file.

There are two ways in which **Ed** may be invoked. The standard way to invoke **Ed** when you are simply editing (creating or changing) a single file is the following command:

Ed myfile

in response to the AmigaDos prompt, where **myfile** is the name of the file you wish to edit. After reading the file into internal memory, **Ed** will respond by printing the number of characters it has read from the file, and it will then wait for your first command. **Ed** normally will not issue a prompt of its own, but you may force it to prompt by using the **P** command explained below. This manner of invoking **Ed** places you in **Ed** in interactive mode, where you issue one command to **Ed** at a time, and **Ed** then executes that command and waits for the next one.

Ed will not automatically make a backup file. There are three ways to handle backups within **Ed**. The first is, when you wish to save the results of your editing changes, write them to a new file rather than the old one by using the **w** command as the following command demonstrates:

w myfile.new

Your original file will remain untouched. A second alternative is to invoke **Ed** with the -b option, thus:

Ed -b myfile

In this case, when you exit your editing session **Ed** will make a backup file called **myfile.bak** that is your original file with its date and time of last modification preserved. If you have no

changes during your editing session, **Ed** will not bother to make the backup file.

Finally, if you neglected to use the **-b** option when invoking **Ed**, you may use the **B** command to instruct **Ed** to make a backup file.

You may also invoke **Ed** in interactive mode without specifying a file name. Your command is simply:

Ed

Ed will respond with a message indicating that it has read nothing in and is waiting for commands. After you enter text, which will be explained below, you may then write this text out to a file by using the **write** command.

Another way of invoking **Ed** is by using the following command:

Ed -c cfile.cmd file_1 file_2 ... file_n

or:

Ed -c cfile.cmd -l files.lst

where **...** indicates that a number of file names may be present.

The **-c** option invokes **Ed** in batch mode, and causes the **Ed** commands contained in the command file **cfile.cmd** to be executed on each of the target files (file_1, file_2, ...file_n). The **-l** option indicates that the following argument is the name of a file containing a list of file names to which the commands are to be applied. **Ed**,s batch mode is discussed in more detail in Section 3.8.

3.2 Interrupting Ed

Ed works under the operating environment of AmigaDOS. If a control-C is typed during an **Ed** session, it is detected by **Ed** and causes an interruption of the current operation. This may be handy if you have invoked a command such as **1,$p** on a long file and get impatient. Typing a control C at this time will interrupt the operation of printing, and return you to command mode, where **Ed** will expect you to type a command. There is no quick, easy method of abandoning **Ed**; in order to quit an editing session, a form of the quit command (**q**) must be invoked.

If you are executing any command other than p when you press the interruption symbol control C you should check the state of your changes to see how much has been done. In order to avoid confusing itself and totally garbling your editing changes, **Ed** will sometimes finish what it is doing before it responds to the interrupt.

3.3 Ed Commands -- Examples

By way of an introduction to **Ed**'s commands, let's look at some examples of common commands you might use. In Section 3.11 a complete and detailed description of all the commands will be provided. The purpose of this section is introduce you to those basic features of **Ed** that will allow you to:

create a file or edit an existing file;
display lines in the file;
add lines to the file;
delete lines from the file;
change lines in the file by substituting one expression for another;
save your changed version of the file; and
exit Ed.

In short, this section contains all the information and examples you need for a complete editing session without involving you in some of the more powerful and esoteric features of **Ed.**

Let's assume that you have just invoked **Ed** in the interactive mode on a new file you wish to create called **myfile.txt** by using the command:

```
Ed myfile.txt
```

Ed will respond with a sign-on message indicating that you are editing a new or empty file.

To begin adding lines to this empty file use the **append** command in the following way:

```
a
This is the first line we are adding to the file.
This is the second line.
How many lines should we add?
Let's add five.
That makes this the last line.
.
```

You have added five lines to our new file. **Ed** must have some way of knowing when to stop adding lines. The way to indicate this is simply to type a period (.) as the first and only character on a line, being sure that it is at the beginning of the line. **Ed** goes from **append** mode back into command mode and is ready to accept more commands.

To display what we now have in our file, type the following command:

```
1,$p
```

Ed responds by printing the following on the screen:

```
This is the first line we are adding to the file.
This is the second line.
How many lines should we add?
Let's add five.
That makes this the last line.
```

In most commands we specify a range of lines followed by the command name. Here the range of lines is:

```
1,$
```

and the command name is:

```
p
```

which means print. The dollar sign ($) is a special sign to Ed that means the last line of a file. So the range of lines we have specified is the first through the last line, and our whole command means "For each line from the first through the last, print it".

Used as a line number, $ means the last line of a file.

The **append** command itself can take a line number, in which case we begin appending lines after the specified line. Thus the command:

```
1a
```

would cause us to begin appending lines after what is now our first line:

```
1a
This is the new second line.
And this is the new third line.
.
```

The result would be the following:

```
1,$p
This is the first line we are adding to the file.
This is the new second line.
And this is the new third line.
This is the second line.
How many lines should we add?
Let's add five.
That makes this the last line.
```

Now there are more than five lines. The next to the last line is

false. Therefore, let's delete it.

In order to delete a single line we must refer to it and use the **delete** command. There are several ways of doing this. First, in a small file like this, we could simply count down to the desired line, which in this case is the sixth. Then our command to delete that line would be:

```
6d
```

and the line would be deleted. There are other ways in which we might achieve this result, however.

An alternative would be to go to that line and simply issue the command:

```
d
```

which means "delete the current line". The concept of the current line is a very important one in **Ed** and will frequently be refered to in the rest of this document. Let's try to understand it now.

At all times in **Ed** there is a current line. (Actually, there are two exceptions to this rule:

(1) when you create a new file with Ed there are no lines in the file, therefore there can't be a current line; and

(2) if you delete all of the lines in your file (e.g., by using the command **1,$d**), there are no lines and therefore no current line).

When you edit an existing file, the first thing that **Ed** does is to read all of its lines into memory and make the last line the current line. Thereafter, which line is the current one depends upon the command you have just executed. If you issue the command:

```
1,3p        /* print lines 1-3 */
```

the current line becomes line 3 (the last one printed). In general, after a command has been executed, the current line will be the last line that the command affected. This is not true for commands that **delete** lines. In that case, the current line is the line after the last line that was deleted. If there is no next line (which would happen if you delete the last line in the file), then the current line is the new last line. **Ed** uses dollar sign (**$**) to mean the last line, and the period (.) to mean the current line.

Used as a line number, . means the current line.

When a command is given without explicitly specifying a line or range of lines, **Ed** assumes that it is to be applied to the current line. Thus both of the following commands mean the same thing:

```
d
```

or:

```
.d
```

Now that you understand what the current line is all about, let's get back to making line 6 the current line and then deleting it. Since the last command was to print all of the lines, you know that the current line is the last line. If you don't know what line you are currently at, however, use the command:

```
p          /* print the current line */
```

to have it displayed. And if you needed to know the line number of the current line, use the command:

```
.=         /* print number of current line */
```

or equivalently:

```
= .
```

and **Ed** will display the line number.

Repeated use of the minus sign (-) allows us to step backwards through the file. Each time we use this command, the previous line is printed and becomes the current line. So to back up to line 6 and delete that line our series of commands is simply:

```
-
Let's add five.
d
```

to check the result:

```
1,$p
This is the first line we are adding to the file.
This is the new second line.
And this is the new third line.
This is the second line.
How many lines should we add?
That makes this the last line.
```

Now the fourth line is false, it says it's the second line. You can delete it or leave the line and reword it so that it is a true statement. Let's do the latter since it will allow us to

use a new command. Once again the problem of determining which line number to give to the command exists. You could simply count and determine that it is the fourth line that needs to be changed, or you could use the minus sign (-) again and back up to it. Let's use another technique that will prove to be quite valuable in large files, you could search for the line and make it the current line.

What you search for is a word or an expression in the line. Since the word **second** occurs in that line, let's search for that:

```
/second/
This is the new second line.
```

Ed displays the the first instance of the word second.

Notice that the previous command was to have **Ed** print all of the lines in the file. So the current line then became the last line printed -- i.e., the current line was the last line in the file. The next command was:

```
/second/
```

which means search forward until you find a line containing the expression **second**, print that line, and make it the current line:

```
/pattern/ means search forward until you find a line in
which pattern is matched, print that line and make it
the current line
```

In **Ed**, all searches take place with wrap-around. What this means is that instead of starting at the line after the current one and continuing to the end of the file, we start at the line after the current one, continue to the end of the file, wrap around to the top (line 1), and continue through the line immediately before the current one.

Since the search started at the last line, **Ed** looked first at line 1, then line 2 where it found the word **second.** That is not the line you wanted, so you need to repeat the command. Use the following command:

```
//
And this is the second line.
```

The command // searches forward for the last pattern you searched for. Since the current line was line 2, the search found the proper line.

The line can be made true by changing **second** to **fourth** by using the following commands:

```
.s/second/fourth/
1,$p
This is the first line we are adding to the file.
This is the new second line.
And this is the new third line.
And this is the fourth line.
How many lines should we add?
That makes this the last line.
```

The first command means "at the current line", **.**, substitute **s** for the first occurrence of the pattern **search** the pattern **fourth.** The current line specifier is not really needed since in the absence of any specifier **Ed** assumes that we you mean to act on the current line. In a later section, the patterns you may ask the **substitute** and **search** commands to act on or look for may be quite general and complex; this is part of what makes **Ed** such a powerful tool.

You have seen how to use **Ed** to create a new file, how to append lines to the file, how to delete lines, how to step through a file, and how to search for a line. These facilities alone will allow you to create and modify files of your own. Now you need to know how to save the version of the file you have modified.

Ed does not work on the file directly. Instead it reads the file entirely into memory and then modifies the image of the file it has in memory. This means is that if you decide you don't really want the changes you have made, all you need to do is quit **Ed** and your original file will remain untouched. To save your changes use the following command:

```
w        /*  write current file image to  disk  */
```

Ed will save the current version of your file on the disk using the current file name. The current file name is the name of the file used when you invoked **Ed.** If you did not use a file name, but said only **Ed** then **Ed** will regard an attempt to save the current file as an error. You should use:

```
w newfile
```

to save the work you have done in the file **newfile** (where newfile is the file name you choose). When **Ed** writes out to a file it prints on the screen the number of characters it has written out.

After saving your work with the write command exit **Ed** by using the quit command. The last few commands and the responses will look like:

```
w       /* save to disk under current file name */
205     /* number of characters written out */
q       /* quit Ed */
1>      /* AmigaDOS prompt */
```

You now know enough of the basics of **Ed** to use it effectively. The other sections will explain additional features and commands of **Ed**, and will go into more detail on the commands covered in this section.

3.4 Special Symbols

The purpose of this section is to introduce you to the set of symbols that have special meaning to **Ed** and to which **Ed** responds in special ways. Many of these are discussed in detail in the **Grep** Section, where the notion of a **pattern** is explained and illustrated. In this section we shall introduce the special symbols (simply to make the user aware of them), and then examine the use of those symbols that are special when used in line number references.

There are a number of special symbols that **Ed** interprets in special ways. Some of these symbols are regarded as special when they are used in a line number specifier, while others are regarded as special when they appear in a pattern or a replacement string. All contexts considered, the following symbols are regarded as special by **Ed** in one way or another:

SYMBOL	SPECIAL MEANING(S)
.	current line "any single character" pattern
$	last line "end of line" pattern
'	label indicator
-	subtraction operator character class specifier
+	addition operator closure operator in patterns
*	closure operator in patterns
/	forward search pattern delimiter
?	backward search pattern delimiter
\	escape character
[ ]	character class delimiters
^	"beginning of line" pattern
!	negation operator in patterns
&	previously matched pattern (in replacement string only)
^C	cancel current input line or interrupt a command

A good principle to keep in mind is this:

If a command does not behave as you expect it to, this may be because you have used a special symbol without intending to have it interpreted in its special way. If so, you may need to escape the symbol to achieve your desired result with the escape character (\).

Let's look now at some of these contexts and the symbols that are regarded as special within them.

A line specifier is any expression we may use to designate a line to **Ed**. The usual way of designating a line, of course, is to use its line number; and so we might use the symbol **5** to designate the fifth line, as in the command:

```
5p          /* print the fifth line */
```

There are other ways to designate a line as well. For example, we could use either **4+1** or **7-2** to designate the fifth line as well. And we have already seen that the special symbols period (.) and dollar sign ($) may be used to designate lines (the current line and the last line, respectively). Therefore, we should begin to see that the class of line specifiers is much broader than the class of line numbers.

We already know that period (.) means the current line and that dollar sign ($) means the last line. But the minus sign (-) and plus sign (+) may be used in a special way as well. For example, the command:

```
---,+++p
```

means "print all of the lines from the third one back from the current line through the third one following the current line", and will cause a total of seven lines (including the current one) to be printed. This is equivalent to:

```
.-3,.+3p
```

The following command means "back up 5 lines":

```
-----
```

More precisely, all this command does is to refer to a line, (the fifth one back from the current line), and **Ed** takes the lack of an explicit command to be an implicit **print** command. The same effect could be gained with the command:

```
.-5
```

which refers to the same line in a different way.

Note that you can cancel the current command by typing **control C.** That is, you don't need to backspace all the way to the beginning of the line. **Ed** will respond by placing the cursor at the beginning of the next (blank) line and waiting for a new command.

These examples adequately cover the use of special symbols in line number specifiers. Before you read the next section on patterns and searching you should first read the **Grep** Section for a full explanation of patterns.

3.5 Patterns and Searching

The purpose of this section is to provide you with examples for constructing patterns and searching a file. There are two ways of searching for a pattern: forward, or backward. To search forward for a pattern, use the command:

/pattern/

and to search backward use:

?pattern?

where **pattern** is whatever pattern you wish to search for. In either case the search takes place with wraparound. In the case of forward search this means that if the pattern is not found before the end of the file is reached, the search continues by going to the top of the file (line 1) and continuing until the current line is encountered. In the case of backward search, if the pattern is not found and line 1 is reached, then the search continues by going to the bottom of the file and searching backwards until the current line is encountered.

If no line is found, then **Ed** displays a question mark (**?**) indicating that the search has failed. Otherwise, the first line in which the given pattern occurs is displayed and becomes the current line. A question mark will also be displayed if the pattern is ill-formed. This can happen, for example, if you have used a special character in the pattern in an inappropriate way, or if you forgot to escape a special character by preceding it with a backslash (\). If you are puzzled by **Ed**'s response to your search request, use the h command to display the latest error message. This may tell you what you did wrong.

Let's look at some examples. To search for a line that begins with the expression "int", use the command:

/^int/

and to search for such a line that also ends with a semicolon, use:

```
/^int.*;$/
```

In this latter pattern, four special characters have been used. The carat (^) anchors searching to the beginning of a line. The sequence ".*" represents zero or more single characters, which is just another way of saying "any string at all, including the null string". The dollar sign ($) ensures that the semicolon (;) must occur at the end of the line.

Suppose you wanted to search for a line that contained the C language comment delimiter string "/*". The following command would not work:

```
//*/
```

It would not work for two reasons. First, when **Ed** sees the second forward slash (/) it will believe that it is being asked to search for the last specified pattern. That is, it will think it is being given the command:

```
//
```

Ed will interpret this as a request to re-search for the last search pattern specified. The final "*/" on the line will be taken as a parameter to the re-search, and **Ed** will return with a question mark (?).

In order to search for the string "/*" you must escape each of the characters so that **Ed** will not interpret them as special characters. Thus, your command should be:

```
/\/\*/
```

In general, you should remember that:

If you don't want a character to be interpreted in its special sense in a pattern, you must escape that character with a backslash (\).

Don't overlook the use of character classes in your search patterns. For example, you may want to search for occurrences of "the" and also of "The". The command you want is:

```
/[Tt]he/
```

Or perhaps you also want to search for "these" and "These" as well. Then the command:

```
/[Tt]he[s]*[e]*/
```

should do the trick. If you don't immediately see how these patterns work, go back to the **Grep** Section and examine how **Grep** treats them.

Once you have specified a search pattern, you may search for the next occurrence of the pattern simply by using the search delimiters with no pattern between them. Thus, "//" searches forward for the next occurrence of the last specified pattern, and "??" searches backward for the next (last?) occurrence of the pattern.

3.6 Substitution

This section assumes that you have read the **Grep** Section; you should do that now if you have not already done so.

The basic form of the substitution command is:

x,ys/pattern/replacement/

where **x** and **y** are line specifiers, **pattern** is a pattern as defined in the **Grep** Section, and **replacement** is the replacement string you wish substituted for **pattern.** The command means:

In lines x through y, replace the first occurrence of pattern in each line with an occurrence of replacement.

In the above example, the forward slash (**/**) has been used to delimit the pattern string from the replacement string, but any other character could be used as well. For example, all of the following are equivalent:

5,7s/int/short/
5,7s.int.short.
5,7s!int!short!

This is a very handy feature. Suppose the string you want to replace contains the forward slash. Suppose you want to replace:

5/9

with:

5%9

then the command:

s/5/9/5%9/

will generate an error message since **Ed** takes:

s/5/9/

to be a substitution command, and interprets the remainder of the line as unintelligible information. There are two ways to accomplish your goal here. The first is to escape the slash that you want replaced so that **Ed** will interpret it as a literal character rather than a delimiter. Therefore use the following:

s/5\/9/5%9/

The other, and somewhat easier, alternative is simply to use another delimiter:

s.5/9.5%9.

The substitution command will replace the **first** occurrence of its pattern (in each line in the given range of lines) with its replacement string. But you may cause it to replace **all** occurrences of the pattern by appending a **g** (for global) to the command as in:

1,20s/malloc/getmem/g

which will replace **every** occurrence of **malloc** with an occurrence of **getmem** in lines 1 through 20.

In order to immediately see the results of your substitution you may append a **p** (for print) to a substitution command, so that:

3,7s!first!last!p

will perform the indicated substitution and print the resulting line. Likewise:

1,20/malloc/getmem/gp

will perform the indicated substitutions and print each line in which such a substitution was made. (It will not print any lines in the specified range in which the substitution was not made.) Note that when both g and p are appended to a substitution command they must occur in that order.

All of the examples of substitutions we have given so far have involved simple strings as the pattern. But the pattern may of course be any regular expression pattern as described in the **Grep** section. For example, you might wish to replace every sequence of spaces and tabs in a file with a single space. The command to do this is:

1,$s.[\t]+. .g

which we should now recognize as meaning:

In every line (i.e., the first through the last lines) globally substitute a single space for each occurrence of a sequence of one or more spaces and tab characters.

As noted above in the section concerning special symbols, the ampersand (&) is special when it occurs in the replacement string. In this case, it stands for the string that matched the pattern specified to be replaced, and that string is then substituted in the replacement string where an ampersand (&)

occurs. For example, suppose you have a file containing a number of lines which are names of files such as:

```
lcl dfl:text -i/lc/
```

and you want to change these into:

```
lcl >dfl:tst/text.err dfl:text.c -i/lc/
```

The substitution commands you would use are:

```
1,$s/dfl:/:/
1,$s/:[a-z]+/>dfl&.err dfl&.c/
```

The first of these deletes the c in front of each occurrence of colon (:). The second then replaces each sequence of letters that is preceded by a colon (:) with a more complex string of the form:

```
dfl_______.err dfl:_______.c
```

where the blanks are filled in with whatever string beginning with a colon (:) was matched in that line. Such combinations of substitution commands as these may be used to make very complicated substitutions in a file.

It is the ability to search for and replace regular expression patterns that makes **Ed** a more powerful editor than most other editors available for microcomputers.

3.7 The global Command

Another sophisticated feature of **Ed** is the **g** (global) command which allows you to pick out a set of lines, and then apply a sequence of commands to each of those lines. For example, in a source file, suppose there are a number of lines in which the function **getmem()** is called, and you want to put a **printf()** statement before each one of these; you could search for each one in turn and repeatedly insert the **printf()** statements manually. Whereas the single command:

```
1,$g/getmem/i\
printf("About to call getmem\n");\
.
```

will accomplish this much more easily.

What this command means is: "At each of the lines in the range from line 1 to the last line that match the pattern **getmem**, insert the line, **printf** ("About to call getmem\n");."

You could simply have each line displayed and its line number printed with the command:

```
1,$g/getmem/p\
.=
```

which means: "At each of the lines in the range from 1 to the last line that match the pattern **getmem**, print that line, then display the number of the current line. The general form of a global command is:

```
i,jg/pattern/cmdline1\
cmdline2\
     .
     .
     .
cmdlinen\
cmdlinen+1
```

where **i** and **j** are line specifiers (including patterns), **g** is the global command name, **pattern** is the pattern used to identify lines to which the list of commands are to be applied, and **cmdline1, cmdline2, ..., cmdlinen+1** are command lines. Note that each command line in the list except the last must end with a backslash (\) to indicate that more are to follow. If there is only one command line in the list, as in:

```
i,jg/pattern/cmdline
```

then of course no backslash is needed.

In executing a global command the first step is to mark every line in the specified range that matches the given pattern. Then for each such line the command list is executed with a period (.) (the current line) initially set to that line. The **g** command itself is not permitted to be in the command list. (An error message will be generated and execution of the global command will halt if a g command is encountered embedded in another **g** command.)

Notice that a command line in the command list may be either a command (such as **a, i, d, p,** etc.) or a line of text to be added, if the previous command was either **a, i,** or **c.** Remember that **a, i,** and c commands must be terminated with a period (.) in column 1 alone on the line. However, in a global command list the line:

```
.\
```

will be recognized as an indication that the current **a, i,** or **c** command is to be terminated and that another command is to follow. A period (.) by itself on a line indicates termination of the current **a, i,** or **c** command and also (since it is not

followed by a backslash) of the global command in which it occurs. Thus the command:

```
20,35g/getmem/i\
printf("About to call getmem\n");\
/* Here is the getmem call */\
.\
.+1\
a\
printf("After call to getmem\n");\
.
```

first identifies each line in the range 20-35 containing the string getmem. At each such line, a **printf()** call and a comment are inserted prior to the line. Then the current line is stepped forward one line (so that the current line becomes the one containing **getmem**). A **printf()** call is then appended to that line. The result is that (within the specified range) each line containing **getmem** is bracketed with **printf()** calls, the first of which is followed by a comment.

If the period (.) terminating an **a**, **i** or **c** command would be the last line in the command list (as it is in the previous example), then it may be omitted. Thus the previous example is equivalent to:

```
20,35g/getmem/i\
printf("About to call getmem\n");\
/* Here is the getmem call */\
.\
.+1\
a\
printf("After call to getmem\n");\
```

As you can see, rather complicated global commands may be constructed. However, this is quite a useful device in even its simplest form. For example, suppose you insert debugging **printf()** statements beginning in column 1. Then when you are finished using them to debug your code, they may easily be stripped from the file with the command:

```
1,$g/^printf/d
```

A certain amount of experimentation will allow you to become familiar with the power of the global command.

The command **v** is the **inverse global command** and is identical to **g** except that it operates on lines that do not match the specified pattern. Neither **g** nor **v** may appear in the command list of a **g** or **v** command.

3.8 Batch Mode

When you are maintaining a number of text files (whether they are programming language source files or documentation files) it is often the case that you need the very same change or series of changes to be made in each of the files. The usual approach is to invoke an editor on each file in turn, and make the required changes. This is quite tedious since you are making the same motions over and over again. It would be simpler to enter your changes once and have the editor take care of applying them to the set of files. **Ed** allows you to do this by means of its batch mode.

To invoke **Ed** in its batch mode, use one of the commands:

```
Ed -c cmdfile file_1 file_2 ... file_n
```

or:

```
Ed -c cmdfile -l filelist
```

In both cases the **-c** option identifies the next argument as the name of an **Ed** command file. A command file is simply a list of **Ed** commands as you would type them while in **Ed**'s interactive mode. In its batch mode, **Ed** will read this list of commands in sequence and apply each of them in turn to the named target files. In the first case above, the target files to which the commands are to be applied are named explicitly on the command line (as **file_1**, **file_2**, and so on). In the second case the **-l** option identifies the next argument as the name of a file consisting of a list (one per line) of file names to which the commands are to be applied. Now let's consider an example.

Suppose you have the files: search.c, io.c, sub.c, lines.c, pat.h, and grep.h. In each of these files you want to make the following changes:

(1) replace every occurrence of **malloc** with **getml**;
(2) replace every occurrence of **int data** with **short data**;
(3) at the top of each file, insert the comment: **/* Revision 1.7 */**.

First, create a command file containing the necessary commands. Let's call the command file **changes.cmd**. **changes.cmd** will look like:

```
1,$s/malloc/getml/g
1,$s/int data;/short data;/g
li
/* Revision 1.7 */
.
w
q
```

Since you might be making other changes to these same files in the future, also create a **targets file** that lists each of the files you want modified. Let's call this file **targets**, and it will look like:

```
search.c
io.c
sub.c
lines.c
pat.h
grep.h
```

To have the changes applied to your files simply use the command:

```
Ed -c changes.cmd -l targets
```

As **Ed** makes the changes on each file it will echo (to standard output) each editor command it is executing. This output may be redirected to a file in the usual AmigaDos manner for later examination:

```
Ed >changes.out -c changes.cmd -l targets
```

Rather than using a file of target names you could also have simply listed the targets on the command line as:

```
Ed -c changes.cmd search.c io.c sub.c lines.c pat.h grep.h
```

In such a command line, **Ed** also will recognize the usual wildcard characters of AmigaDOS. Thus to apply our changes to all of the files with extension .c and .h in the current directory, we could invoke the command:

```
Ed -c changes.cmd #?.c #?.h
```

If you are more comfortable with MS-DOS file names and wildcards, you can use the -m (for MS-DOS file name patterns) switch when invoking **Ed**:

```
Ed -m -c changes.cmd *.c *.h
```

In batch mode **Ed** proceeds under its own control, it is wise to turn on the printing of **Ed** error messages in case it cannot succeed in making the changes it is instructed to make. Accordingly, the command file would then be:

```
h
1,$s/malloc/getml/g
1,$s/int data;/short data;/g
1i
/* Revision 1.7 */
.
w
q
```

In using batch mode there are several points to keep in mind. First, when you use either the **a** or **i** commands to add text to the file, make sure that you don't forget the period (.) that will tell **Ed** that input has been terminated. Otherwise, **Ed** will continue to assume that the lines it finds in the command file following the a or i are intended as input rather than commands. In such a case **Ed** will hang because it will get to the end of its command file and will be waiting for additional input. You will then have to type a period (.) in order to force **Ed** to exit its insert or append mode.

Second, remember that by default **Ed** does not make backup files. If you want backup files made in batch mode, don't forget to use the **-b** option on the command line when you invoke **Ed.**

Finally, when in batch mode **Ed** treats q and **Q** equivalently as unconditional quit commands. This could be useful in preventing **Ed** from hanging if something should go wrong.

3.9 Error Messages

There are a multitude of error messages and warnings that **Ed** will display should you make a mistake. These are intended to be self-explanatory and therefore will not be reviewed here. Messages referring to patterns (such as improper pattern, and so on) are discussed in the **Grep** Section. You should be familiar with this guide prior to using the search and substitution commands of **Ed.**

By default, **Ed** will respond to an error with a cryptic question mark (?) comment. A brief description of the problem may be retrieved by means of the h ("print most recent error message") command. Those who find such terseness distressing should use the **h** (turn on/off printing of error messages) command immediately after entering **Ed.** Thereafter, the question mark (?) will be followed by a display of the appropriate message.

Generally, an error simply causes **Ed** to cancel the current attempt (or to refrain from any attempt at all) and give you a new opportunity to rephrase your request. However, there are some circumstances in which **Ed** will get partially through its action before detecting a problem. In particular, any "Out of Memory" message is serious. Do not despair. At worst you can simply save your changed file (use **w** to write it to a file of your choice) and lose little, if any, of your work. Alternately, you can use w with a range of lines to save your work in pieces so that you may later edit the pieces without running out of memory. For example, the commands:

```
1,500w temp1
501,$w temp2
```

will save the current version of the file in two files (temp1 and temp2).

Running out of memory should not be a frequent problem. **Ed** uses no fixed size tables, but instead dynamically allocates space as it is needed. You are therefore limited only by the amount memory on your machine that is accessible through the operating system.

In the direst of circumstances you may see an "Internal error" message. This means that something rather unexpected and disastrous has gone wrong inside of **Ed.** Save what you can (to a temporary file, in order to avoid writing over the original file) and report all of the details to Lattice.

3.10 Size Limitations

As mentioned above, **Ed** does not use any statically sized structures for storing its lines. However, there are some size limitations that the user should keep in mind.

The line buffer used to hold lines of text input either from the keyboard or from a file is 512 bytes long. If a line (including the terminating NULL character) is longer than this, the line will be truncated and a warning displayed to the user.

The buffer used to hold the result of a substitution in a line is three times the size of the line buffer, or 1536 bytes. This should provide ample room when substituting in a line.

The buffer used to hold a command line is 256 bytes, and it is difficult to imagine how one might exceed this limit. (Note that a global command may contain a list of command lines. The size of the command line buffer restricts only the size of each individual command line in the global command. It does not restrict a global command to being 256 bytes or less.)

A pattern string (used in the search or substitution commands) may be as long as 80 characters (including the termination character).

A file name by be up to 64 characters, and AmigaDos path names are recognized.

Ed stores its lines as a doubly linked list of structures, each such structure requiring 16 bytes of memory. In addition to this, of course, is the size of the text string for that line.

Finally, **Ed** finds it necessary to allocate additional space from time to time in order to execute various commands such as the search and global commands. Such space is freed when no longer needed, as is any space from lines that are deleted.

3.11 Command Summary

In the summary of commands below, each command appears in the context of a synopsis followed by a description of that command. The synopsis is a generalized canonical form of the command. In the synopsis, **L_1**, **L_2**, and **L_3** are used to denote line specifiers (as described in Section 3.1) the expression **...text...** is used to denote any text that may be entered at the position of that expression; **PATTERN** denotes a regular expression pattern as this is defined in the **Grep** Section; **replacement** denotes any string intended to serve as the replacement of such a pattern; and **FILE** stands for any file name.

SYNOPSIS	DESCRIPTION
!	The command is passed on to AmigaDos for execution.
#	Display a brief command summary on the screen.
L_1=	The number of the specified line is displayed. If L_1 is omitted '.' is assumed.
%	The current line is displayed, set off by single horizontal lines. Above it, the prior five lines are displayed, and below it the succeeding five lines are displayed. The display is bounded above and below by double horizontal lines.
L_1a	The "append" command reads the given text and appends it after the

specified line (which is '.') if L_1 is omitted. If L_1 evaluates to Ø, the text is placed before line 1. '.' is left at the last line appended.

B This command toggles on and off the "automatic backup" feature of **Ed**. By default, **Ed** does not automatically make a backup file when it exits. The **-b** command line option will force a backup to be made. The **B** command will toggle the backup feature to either on or off and will display its status.

L_1,L_2**c** The "change" command first deletes the specified range of lines. The input text then replaces these lines. '.' is left at the last line input or, if there were none, at the line after the last deletion. Either one or both of the line specifiers may be omitted.

L_1,L_2**d** The "delete" command deletes the specified range of lines. Either one or both of the line specifiers may be omitted.

e FILE The entire contents of the buffer is deleted and the specified file is read in for editing.

f FILE The current file name is changed to FILE. If FILE is omitted, the current file name is displayed.

L_1,L_2**g**/PATTERN/command list The "global" command first identifies each line in which PATTERN is matched. Then, for each such line, '.' is set to that line and the given list of commands is executed. If the command list is longer than a single command, each command (or input line, in the case of **a**, **i**, or **c** commands) in the list except the last must end with a backslash (\). An empty command list is equivalent to the "print" command. The command list may not include a **g** or **v** command.

h	Print the most recent error message.
H	This command toggles on and off the automatic printing of error messages. By default **Ed** prints only a question mark (?) when an error is encountered.
L_1**i**	The "insert" command is identical to the "append" command except that the text is inserted before the specified line. Line Ø is not recognized as a proper line in this command.
L_1,L_2**j**	Joins continuous lines by removing the newline characters.
L_1**k**x	Marks the addressed line with the name x, which must be a lower-case letter. Thereafter, the marked line may be specified as 'x (a single quotation mark followed by the letter).
L_1,L_2**m**L_3	The "move" command moves the specified block of line(s) after the line L_3. Ø is a legal value for L_3 and causes the beginning of the buffer. '.' is left at the last line moved. If L_1 and L_2 are omitted, '.' is assumed.
N	This command toggles on and off the printing of line numbers when lines of text are displayed. By default this feature is turned off.
L_1,L_2**p**	The "print" command prints the specified range of lines on the screen. Either L_1, or both L_1 and L_2 may be omitted.
P	This command toggles on and off the **Ed** prompt, an asterisk (*). By default the prompt is turned off.
q	Checks for changes and exits **Ed.**
Q	Unconditionally exits **Ed.**

L_1**r** File

Reads the text of the specified file into the buffer after the specified line (which is '.' if L_1 is omitted).

L_1,L_2**s**/PATTERN/replacement/g
L_1,L_2**s**/PATTERN/replacement/

The "substitute" command searches each line in the given range for those in which PATTERN is matched. In each such line, the first occurrence of PATTERN is replaced with the replacement string. If the terminating g is specified on the command line, then all (non-overlapping) occurrences are simultaneously replaced. Any printable character may be used (rather than the backslash) as a delimiter.

An ampersand (**&**) occuring in the replacement is replaced by the string matching PATTERN in the current line. This special sense of the ampersand character may be suppressed by preceding it with a backslash (\).

A line may be split by substituting a newline character (**\n**) into it.

Either form of the command may be followed (on the same command line) with a p to force the printing of the result(s) of substitution.

L_1,L_2**t**L_3

This is identical to the **m** command except that the specified lines are copied rather than moved. '.' is left at the last line copied.

L_1,L_2**v**/PATTERN/command list

This is identical to the g command except that the command list is applied to all lines in which PATTERN is not matched.

L_1,L_2w FILE	The specified range of lines is written to the given file. If FILE is omitted, the current file is assumed. Either L_1 or both L_1 and L_2 may be omitted. The number of characters written is printed if the command succeeds.

SECTION 4:
Using Splat

4.1 Introduction to Splat

Splat is a **stream editor** for files. Given a pattern, a substitution string and a set of files, **Splat** quickly replaces all occurrences of the pattern with the string in each of the files. The original version of the file is left unmolested (unless the **-o** option is used) and **Splat** will either place the result of its substitutions in a temporary file (with the same root file name, but with a ".$$$" extension) or in a "backup" directory specified by the user.

Splat is especially useful when fairly simple substitutions need to be made in a number of source files. For example, suppose you discover that in all of your C language source files you need to replace all unsigned character declarations with character declarations. The **Splat** command to do this would be:

```
splat "unsigned char" char #?.c
```

Splat is both faster and more convenient than a text editor since:

(1) it does not need to read the entire file into memory while it does its substitutions;

(2) it can be applied once to a set of files rather than invoked repeatedly on single files; and

(3) it will not modify the original version of the file it is working on.

4.2 Using Splat

Splat is invoked in the following way:

splat [-s] [-o | -dPREFIX -m -v] pattern string file1 ... filen

Here **pattern** is a regular expression pattern as described in the Grep Section, and **string** is the string you desire to be substituted for that pattern. Note that **Splat** uses the same technique as does **Ed** in performing its substitutions, and that characters in the substitution string that are special to **Ed** (e.g., **&**) are special to **Splat** in exactly the same way. Similarly, characters that are special when they occur in patterns (e.g., **.**, **[ ]**, **$**, etc.) are special when they occur in the **pattern** argument of **Splat**. Refer to the sections on **Ed** and **Grep** for a complete characterization of these special characters.

The **-s** option forces **Splat** to announce the name of each file as it begins its work on that file, and to announce that no substitutions have been performed on a file if it was unable to find the specified pattern in that file. Without the **-s** flag **Splat** does its work silently.

Only one of the **-o** or **-d** options may be used since they are incompatible with one another. The -o option tells **Splat** to **overwrite** the original version of the file with the new version. With the -d option, the user specifies a directory prefix which indicates the directory into which **Splat** will place its new files (leaving the original versions untouched). Thus the command:

splat -d/bak/ int INT #?.h #?.c

causes "int" to be replaced with "INT" in all files with **.h** and **.c** extensions, and the new versions of these files to be placed in the directory **/bak**. Beware that this command will replace such expressions as "printf" with "prINTf" -- which may not be the desired result. Likewise, the command:

splat -dbak/ int INT #?.h #?.c

will perform the same substitutions, but it will place the new versions of the files in the subdirectory **bak/** of the user's current directory.

If the specified directory does not exist, **Splat** attempts to make it. Should the attempt to create the directory fail, an error message to that effect is displayed to the user.

The **-m** (MS-DOS file name patterns) option will cause interpretation of any wildcards used in file specifications to obey MS-DOS file name conventions. That is, an asterisk (*) for matching zero or more of any characters, a question mark (?) for matching any single character.

The **-v** (verbose) option causes **Splat** to display all lines in which substitutions are made to standard output. No files are created when using this option, and it is intended only to give the user a quick preview of the changes **Splat** will make when it is creating modified files.

4.3 Examples

Since (in the absence of the **-o** option) **Splat** will not overwrite the original versions of your files, it is safe and easy to experiment with the tool. Such experimentation is often wise since a careless specification of the regular expression pattern may result in undesired changes.

Suppose, for example, we wish to go through each of our source files and substitute "INT" for "int" in every declaration of an integer identifier. But we don't want our "printf" calls to become "prINTf". If we know that in each such integer declaration, the expression "int" is preceded and succeeded by a space, then the command to use is:

```
splat -dbak/ " int "  " INT " #?.c
```

Keep in mind that (as in **Grep**) a pattern containing white space must be enclosed in double quotes.

Again, suppose that for each line of the file **comp.bat** on which the expression "lcl" occurs we wish to append the expression **-cc** at the end of the line. Then the command:

```
splat -dbak/ "lcl.*$" "& -cc" comp.bat
```

should do the trick. You may need to refer to the Ed section and the Grep Section to see that the pattern **"lcl.*$"** will match a string beginning with "lcl" and ending with the end of the line, and that the substitution string **"& -cc"** will expand to the matched string followed by a space followed by **"-cc"**.

It is even possible to insert lines in a file with **Splat**, or to break single lines into multiple lines. For example, if the expression:

```
parms(s);usage();exit(1);
```

occurs on a single line of the file **test.c** and you want the function calls to **usage()** and **exit()** to be on separate lines and indented one tab stop, then you should use the command:

```
Splat  parms(s);usage();exit(1)
     parms(s);\n\tusage();\n\texit(1);\n\ttest.c
```

to insert the newline and tab characters.

As you gain familiarity with **Splat** you will see that fairly complicated substitutions may be accomplished with a judicious choice of the regular expression pattern and substitution string.

SECTION 5:
Using Extract and Build

5.1 Introduction to Extract and Build

Extract and **Build** are two simple utilities that are useful for constructing batch files for AmigaDOS. **Extract**, as its name indicates, extracts file names from the current directory. **Build** causes the insertion or interleaving of lines of text in a given file. Used together they comprise a powerful tool for the automating of complex tasks under AmigaDOS. The resultant **batch** file can then be executed by invoking:

Execute batchfile

from the AmigaDOS CLI prompt, where batchfile is the name of the batchfile created from the extract and build utilities.

5.2 Using Extract

The way in which you invoke **Extract** is:

extract [-r | -b] [-n] pattern [pattern ...]

where the **-r**, **-b**, and **-n** flags are optional, and each **pattern** is a file name, possibly containing wildcard characters. For example, the command:

extract #?.c

will result in each file name with a .c extension being printed on your screen (one name per line). To direct this list to a file, simply use:

extract >names #?.c

and the file names will appear in the file **names**.

Note, the commandline redirection of output is handled by AmigaDOS, not by **Extract**, and hence must appear as the second parameter on the command line.

The **-r** (root) flag indicates that only the root portion of the file name is to be output. That is, any prefix (such as **'df0'**: or /docs/) or suffix (such as .c) is stripped from the name. The **-b** (basename) flag is similar to the **-r** flag, but leaves the extension on the name.

The **-n** (no sort) prevents the list of file names from being sorted. By default, the file names are sorted in alphabetical order.

5.3 Using Build

When **Build** is invoked as:

```
build >batch.bat listfile
```

listfile is assumed to be a file consisting of a number of lines of text. Typically, each line will be the name of a file if you are using **Extract** and **Build** to construct batch files. However, these utilities may be used for other purposes as well. For the sake of simplicity, let's call each line in **listfile** a **name**.

Build will wait (without a prompt, because it is reading from standard input) for input from the keyboard. Each line of input is thought of as a **model** from which to construct lines that are printed to the output file. The end of your input is indicated by typing a control \ and carriage return as the last line. (If input is redirected from a file, this is detected as end-of-file.)

A **model** is a sequence of text lines in which the characters exclamation point (**!**) and forward slash (**/**) have special interpretations. Each model line in which an exclamation point (**!**) appears is repeated for every name in the name list, and the corresponding name replaces the character. Within any model line, the forward slash indicates that a newline character is to be generated at that point. This feature allows multiple output lines to be generated for each name. To use a literal forward slash, you must escape it with a second one, hence **//**.

Model lines are read one at a time (from standard input), and the name list file is rewound (i.e., you start back at the top of the file) after each sequence of output lines is generated from a given model line.

5.4 Examples

For the first example, let's assume that you want to use an upload/download program to transfer files from an Amiga to a remote minicomputer. Further assume that the upload/download program expects to read a file which contains text blocks of the following format:

```
Name of file on Amiga to be sent
Action to be performed (in this case, "lsend")
Name of file on remote machine to be created
The word "end" to indicate the end of this action
```

Suppose the current directory contains the files: **stdio.h**, **ctype.h**, **fcntl.h**, **main1.c**, **lst.c**, **io.c** and **globals.c**. Use **Extract** and **Build** to construct the file to be used as input to

the upload/download program. The first step is as follows:

```
extract >names #?.h #?.c
```

This produces in the file **names** the list:

```
ctype.h
fcntl.h
stdio.h
globals.c
io.c
lst.c
mainl.c
```

then use the command:

```
build >upload names
```

and type in from the keyboard:

```
!/lsend/!/end
```

followed by a control \ and carriage return as the last line.

This produces the file **upload** which looks like:

```
ctype.h
lsend
ctype.h
end
stdio.h
lsend
stdio.h
end
fcntl.h
lsend
fcntl.h
end
globals.c
lsend
globals.c
end
io.c
lsend
io.c
end
lst.c
lsend
lst.c
end
mainl.c
lsend
```

```
mainl.c
end
```

As another example, let's suppose that you want to construct a batch file for compiling the files mainl.c, lst.c and io.c.

After the compilation is performed, copy the objects to drive dfl: and delete them from the current directory. Begin with:

```
extract >names -r mainl.c lst.c io.c
```

which creates names containing:

```
io
lst
mainl
```

then invoke **Build** as follows:

```
build >comp.bat names
```

type the following:

```
cd source
lcl ! -i//lc///lc2 !
copy !.o dfl:!.o
delete !
```

Terminate input with a control \. This results in the file comp.bat as follows:

```
cd source
lcl io -i/lc/
lc2 io
lcl lst -i/lc/
lc2 lst
lcl mainl -i/lc/
lc2 mainl
copy io.o dfl:io.o
copy lst.o dfl:lst.o
copy mainl.o dfl:mainl.o
delete io
delete lst
delete mainl
```

Although the examples above are quite simple, they should serve to illustrate the basic features of **Extract** and **Build** and to encourage you to experiment with more complicated applications.

SECTION 6:
Using the Files Command

6.1 Introduction to the Files Command

The **Files** utility is a powerful program that allows you to search for, copy or erase files or directories. In some respects it is similar to the UNIX find utility, but is easier to use. In addition to the file searching ability of find, it has much of the functionality of the UNIX cpio and rm commands in its ability to copy, move or remove entire directory structures. The utility may be used to search one or more directories;

looking for files that have a specified name;
match a specified pattern;
are of a specified type (directory or normal file);
have modification dates within the past specified number of days;
are newer than a specified file;
older than a specified file;
smaller than a specified size; or
are larger than a specified size.

It is particularly handy if you know that you have a file somewhere on your disk but have forgotten exactly where it is. Even if you recall only a part of the file's name, **Files** can find the file for you. In addition, **Files** will allow you to copy entire directory structures from one subdirectory to another or from one drive to another. The utility also supports a recursive erase feature that allows you to remove an entire directory structure and all of its files with a single command.

6.2 Using the Files Command

The general form of the **Files** command is:

files [options] dir1 dir2 ...

where **dir1, dir2, ...** are directory names and **options** is a sequence of options described in section 6.3. For example, to find all files in the directory **/lc**, use the command:

files /lc

and **Files** will respond with a listing of all files in the /lc directory such as:

```
/LC/LC1
/LC/LC2
/LC/STDIO.H
    .
    .
    .
```

Similarly, the command:

```
files /lc .
```

will generate a listing of all files in either the /lc directory or the current directory.

Note that **Files** searches not only the specified directories but that it searches (recursively) all subdirectories of the specified directories, all subdirectories of those subdirectories and so on.

6.3 Options

Usually you do not need a listing of every file in a directory, but only a listing of certain files in that directory. The **Files** utility provides a number of options to accomplish this goal. Except where specifically stated otherwise these options may be used in conjunction with one another.

-name option The **-name** option allows you to search for files that match a specified name. For example, the command:

```
files -name grep.h .
```

will look for a file whose name is grep.h in the current directory.

Note that the case of filenames is not considered by AmigaDOS. Likewise, **Files** is compatible with this usage, and will detect files whether they appear as GREP.H, grep.h, Grep.h, etc.

Likewise, the command:

```
files -name #?.c tools/tmu
```

will search for all files with .c extensions in the subdirectory tools/tmu of the current directory. The command:

files -name /tmu /tools

will list all files found in any subdirectory (sub-subdirectory, etc.) called **tmu** of the directory /tools.

The file name specified with the **-name** option may contain the standard AmigaDOS wildcard characters and these will be interpreted in their usual sense.

To use **Files** in the MS-DOS compatibility mode, you must invoke the **-m** switch from the command-line. This will allow the use of MS-DOS wildcard patterns in your file specification. An example:

files -m -name *.c .

would search for all files with the extension .c in the current directory.

-pat option The **-pat** option is similar to the **-name** option with the exception that the pattern specified is interpreted as a regular expression pattern in the manner of **Grep**. Thus the command:

files -pat text[Ø-9] /docs

will search the directory /docs for any file whose name contains the word text followed by a decimal digit. In using the **-pat** option you must remember that a number of characters (e.g., ., +, *, etc.) are interpreted in a special sense. See the **Grep** Section for more details on these patterns.

Again, like the **-name** option, this option translates the specified pattern to upper case prior to beginning its search.

Note that the **-pat** and **-name** options are incompatible with one another.

-type option The **-type** option allows you to restrict you search to files of a given type. The command:

files -type f . /docs

will list only the normal files in the current

directory and the /docs directory. The command:

files -type d . /docs

will list only the (sub)directories in these directories (and, recursively, in their subdirectories).

Of course the **-type f** and **-type d** options are incompatible with one another.

-days option If you want to search the current directory and the directory /x for files which have been created or modified within the last five days, use the command:

files -days 5 . /x

Any positive (decimal) number may be used as a parameter to the **-days** option.

-newer option Do you want to find all files in the subdirectory bugs newer than the file /src/test.c? Simply use the command:

files -newer /src/test.c bugs

-older option This is similar to the **-newer** option except that only files older than the one specified are selected.

-size option This option allows you to search for files which are at least as large as a specified size. For example, to search for files that are at least 20,000 bytes in size in the current directory, use the command:

files -size 20000 .

-copy option This option allows you to copy files or directories from one place to another. The command:

files -copy dfl: /vms

will copy to drive dfl: all files in the directory /vms (and all files in subdirectories of /vms subdirectories, etc.). The command will make any new directories or subdirectories necessary at the destination and copy all files. Similarly, the command:

files -name #?.c -copy /test/tmp /lc/bugs /link/tests

will copy all files with .c extensions that are anywhere under the directories **/test/tmp** or **/link/tests** to the directory **/test/tmp** (making subdirectories as necessary). Note that the target directory must exist in order for the copy to take place.

In the above example, the directory **/test/tmp** must already exist or **Files** will complain that it cannot create the new (copied) versions of the files.

With this option, **Files** provides much of the functionality of the UNIX tar or cpio commands. Used with the **-erase** or **-rerase** options, the **-copy** option may be used to move directory trees since the copy is done prior to the erasure.

-erase option
-rerase option These options cause **Files** to erase the files that are selected. The **-erase** option will not descend into subdirectories, but will erase files only at the highest level of the specified directory. On the other hand, the **-rerase** option will recursively descend into subdirectories and remove all files that are selected. Thus, for example, the command:

files -erase -name #?.c /lc

will remove all **.c** files from the directory /lc whereas the command:

files -rerase -name #?.c /lc

will remove all .c files anywhere under /lc or its subdirectories, subsubdirectories, etc. Finally, the command:

files -rerase :

will remove all files on your current disk drive. This is equivalent to the UNIX rm -rf * command.

-b option
-r option These are identical to the **-b** and **-r** options in **Extract**, producing base and root names of files.

-n option This option forces **Files** to refrain from recursively descending into subdirectories to do its work whether it be finding, copying or erasing.

-v option Normally when erasing files, the **Files** command does its work silently. This option forces it to announce the files and directories it is removing.

Many of the options may be used in conjunction with one another. For example, you may copy and erase files (thus moving them), find files newer than one file and older than another, and so on. If your selection of options is incompatible, **Files** will issue an error message to that effect.

SECTION 7:
A Note on WordStar Files

Certain options available with **Grep**, **Diff** and **Wc** facilitate their use on WordStar files. In particular, the **-p** option filters out high bits and non-printable characters to present the user with a readable display of text in any file created by using WordStar.

Under certain circumstances a user may wish to change a WordStar file into a normal file. For example, one might want to make use of a documentation facility (such as UNIX nroff or Edix). If all of one's documentation is currently in WordStar format such a changeover might be viewed as a formidable task. However, **Grep** allows you to get a jump on such a conversion job.

The command:

grep >newfile -n -p ^ oldfile

will scan the file **oldfile** (assumed to be in WordStar format) and produce the file **newfile** which is in identical format but consists entirely of printable characters. Of course, any information concerning expressions which were in italics or boldface in **oldfile** will be lost when the translation takes place to **newfile.** But virtually all other information will be retained. Note that this includes WordStar control lines (such as **.pa** to force a new page, etc.).

To exclude control lines from your new file, simply use the command:

grep >newfile -n -p ^[!\.] oldfile

This will cause **Grep** to select out of **oldfile** any line that does not begin with a period (.). Thus WordStar control lines will not be selected. Be aware that if you have lines in your file that begin with a period (.) but are not WordStar control lines, these too, will fail to be printed.

INDEX

\, 1-2, 1-13, **3-10**, 3-12, 3-13, 3-18, 3-25, 3-27

" ", 1-3, 1-8, 1-14

!, 1-2, 1-6, 1-14, **3-10**, **3-24**, 5-2

#, **3-24**

$, 1-2, 1-8, 1-13, 3-4, 3-5, **3-10**, 3-13

%, **3-24**

&, **3-10**, 3-27

', **3-10**, 3-26

*, 1-2, 1-6, 1-8, 1-13, 1-15, **3-10**, **3-26**, 6-3

+, 1-2, 1-6, 1-8, 1-15, **3-10**, 6-3

-, 1-2, 1-5, 1-20, 3-6, **3-10**, 4-1
-$, 1-16
-b, **2-2**, 2-6, 3-1, 3-2, 3-25, 5-1, **6-5**
-b Option, 3-22
-c, 1-15, 3-2, 3-20
-c Option, 3-20
-copy, **6-4**, 6-5
-d, 4-2
-days, **6-4**
-e, **2-2**, 2-3, 2-4
-erase, 6-5
-f, 1-15
-l, **2-1**, 2-6, 2-7, 3-2, 3-20
-m, 1-15, 4-2
-n, 1-15, 5-1, **6-6**
-name, **6-2**, 6-3
-newer, **6-4**
-o, **2-1**, 4-1, 4-2
-older, **6-4**
-p, 1-16, 2-1, **2-2**, 7-1
-pat, **6-3**
-r, 5-1, **6-5**
-rerase, 6-5
-s, 1-15, 2-1, 2-2, 4-1, 4-2
-size, **6-4**
-type d, **6-3**, 6-4
-type f, **6-3**, 6-4
-V, 1-15, 1-15, 4-2, **6-6**
-w, **2-2**, 2-6, 2-7

.., 1-2, 1-4, 1-13, 1-14, 3-3, 3-5, 3-8, **3-10**, 3-24, 3-25, 3-27, 6-3, 7-1

/, **3-10**, 3-15, 5-2
//, 3-7, 3-14, 5-2

:, 3-17

;, 3-13

=, 3-6, **3-24**

>, 1-16

?, **3-10**, 3-12, 3-13, 3-22, 3-26
??, 3-12, 3-14

a, 3-3, 3-4, **3-24**, 3-25, 3-26
Addition Operator, **3-10**
Advanced Examples, **1-9**
AmigaDOS User's Manual, 1-16
Ampersand, **3-10**, 3-27
Anchored Regular Expression Match, **1-18**, **1-19**
Anchored Searches, **1-8**
Append Block, 2-4
Append Command, 3-3, 3-4, **3-24**, 3-25, 3-26
Append Mode, 3-3
are_match, 1-17, **1-18**, 1-19
are_smatch, **1-19**
ASCII Character, 1-16, 2-2
Asterisk, **3-10**, **3-26**

b, 1-3, 1-14, 3-2, **3-25**
Backslash, 1-2, 1-13, 1-14, **3-10**, 3-12, 3-13, 3-18, 3-25, 3-27
Backspace, 1-14
Backspace Character, 1-3
Backup, 3-1, 3-2
Backup Command, **3-25**
Backup Directory, 4-1
Backup File, 3-1
Backward Search Pattern Delimiter, **3-10**
Bad character class, **1-20**
Bad pattern, **1-20**
Basename, 5-1
Batch Mode, 3-2, **3-20**, 3-22
Beginning of Line Pattern, **3-10**
Bracket, 1-2, 1-5, 1-14
Build, 5-1, **5-2**

c, 3-25
Can't find file(s), **1-20**
Can't open, **1-20**, **2-6**
Cancel Current Input Line, **3-10**
Carat, 1-2, 1-8, 1-13, 1-14, **3-10**, 3-13
Carriage Return, 1-14, 2-1, 5-2, 5-3
Change Block, 2-4
Change Blocks, 2-3
  Append Block, 2-4
  Change Block, 2-4
  Delete Block, 2-3
Change Command, **3-25**
Change File Name Command, **3-25**
Character Class Delimiters, **3-10**
Character Class Specifier, **3-10**
Character Classes, **1-5**, 3-13
Checksum, 2-2
Closing] not found, **1-20**
Closure Operator in Patterns, **3-10**
Closures, **1-6**
Colon, 3-17
Command File, 3-2, 3-20, 3-21
Command Line, 1-16
Command Mode, 3-3
Command Summary, **3-24**
Concatenation, 1-15
Control, 5-2, 5-3, 5-4
Control C, 3-2, **3-10**
Control Characters, 1-16
Control G, 1-3
Copy Lines Command, **3-27**
cpio, 6-1, 6-5
Current Line, 3-5, **3-10**

d, 3-5, **3-25**
Dash, 1-2
Default, 2-2
Delete Block, 2-3
Delete Command, 3-5, **3-25**
Delete Contents of Buffer Command, **3-25**
Diff, **2-1**, 7-1
Diff I/O Error, **2-6**
Diff Options, **2-1**
Differential File Comparator, **2-1**
Directory, 6-2
Display, 3-3
Display Command Summary, **3-24**
Display Current Line Command, **3-24**
Display Error Message, 3-12
Display Specified Line Number Command, **3-24**
Dollar Sign, 1-2, 1-8, 1-13, 3-4, 3-5, **3-10**, 3-13
Double Quote, 1-3, 1-14

Downloading, 2-3

e, 3-25
EBCDIC Character Set, 2-3
Ed, 1-1, 2-4, 3-1
Ed Command Examples, 3-3
Ed Error Messages, 3-22
Editor Script, 2-2
Edix, 7-1
Empty character class, 1-20
End of Line Pattern, 3-10
End-of-File, 5-2
End-of-String, 1-14
Error Messages, 1-20, 1-22, 2-6
  Bad character class, 1-20
  Bad pattern, 1-20
  Can't find file(s), 1-20
  Can't open, 1-20, 2-6
  Closing] not found, 1-20
  Diff I/O Error, 2-6
  Empty character class, 1-20
  Improper -l specification, 2-6
  Improper hex specification, 1-21
  Incompatible combination of options, 1-21
  Internal Error, 2-7
  Invalid option, 1-20
  Line table overflow, 2-7
  No beginning double quote in pattern, 1-21
  No file arguments provided, 1-21
  No pattern or file arguments given, 1-21
  Not enough memory for ... lines per file, 2-7
  Out of memory, 2-7
  Out of space, 1-21
  Pattern ill-formed: no terminating double quote, 1-21
  Too few arguments to **Grep**, 1-21
  Too many file names:, 2-7
  Unrecognized option, 2-7
Escape, 1-2, 1-3, 1-13, 1-21, 3-15
Escape Character, 3-13
Escape Indicator, 3-10
Examples, 5-2
Exclamation Point, 1-2, 1-6, 1-14, 3-10, 5-2
Exit **Ed** Command, 3-26
Extract, 5-1
Extract File Names, 5-1

f, 3-25
File Name Patterns, 1-15
Files, 6-1
Files Options, 6-2
find Utility, 6-1
Flags, 1-2, 1-15

Forward Search Pattern Delimiter, **3-10**
Forward Slash, **3-10**, 3-14, 3-15, 5-2

g, 3-19, **3-25**, 3-27
g Command, 3-18
Global, 3-17
Global Command, **3-25**, **3-27**
Global Regular Expression Search, 1-1, **1-13**
Grep, **1-1**, **1-13**, 1-17, 7-1
Grep Command, 1-2
Grep Examples, **1-1**

h, 3-12, 3-22, **3-26**
Hex Character, 1-21
Hex Number, 1-3, 1-14
High Bits, 7-1

i, 3-25, **3-26**
I/O Buffer, **2-2**, 2-6
Ignore Case, 1-16
Implementation Issues, **2-5**
Improper -l specification, **2-6**
Improper hex specification, **1-21**
Incompatible combination of options, **1-21**
Insert Command, 3-25, **3-26**
Insert Lines of Text, 5-1
Interactive Mode, 3-1, 3-2
Internal Error, **2-7**
Interrupt a Command, **3-10**
Interrupting **Ed**, **3-2**
Introduction to **Build**, **5-1**
Introduction to **Diff**, **2-1**
Introduction to **Ed**, **3-1**
Introduction to **Extract**, 5-1
Introduction to **Grep**, **1-1**
Introduction to **Splat**, **4-1**
Introduction to the **Files** Command, **6-1**
Introduction to **Wc**, **2-1**
Invalid option, **1-20**
Inverse Global Command, 3-19
Invoking **Ed**, 3-1, 3-2

j, **3-26**
Join Lines Command, **3-26**

k, **3-26**

Lable Indicator, **3-10**
Last Line, 3-5, **3-10**
Lattice Line Editor, 1-1, 2-2, 2-4
lc.lib, 1-12
Left Bracket, 1-2, 1-13, 1-14, **3-10**

Line Buffer, 3-23
Line Number, 1-15
Line Number Specifier, 3-9
Line Specifier, 3-11
Line Table, 2-6, 2-7
Line Table Overflow, **2-7**
Linefeed, 2-1

m, 3-**26**, 3-27
Manual Pages, **1-12**
Mark Lines Command, **3-26**
Metacomco Linker, 1-12
Minus Sign, 1-5, 1-**20**, **3-10**
Model, 5-2
Move Block Command, **3-26**
Move Lines Command, 3-27
MS-DOS, 1-16

n, 1-3, 1-14, **3-26**, 3-27
Negation Operator, 1-6
Negation Operator in Patterns, **3-10**
Newline Character, 3-27
Newline, 1-14, 4-3
Newline Character, 1-3, 5-2
No beginning double quote in pattern, **1-21**
No file arguments provided, **1-21**
No pattern or file arguments given, **1-21**
No Sort, 5-1
Non-Graphic Characters, 1-14
Non-Printable Characters, 1-16, **2-2**, 7-1
Not enough memory for ... lines per file, **2-7**
Not Sign, 1-14
nroff, 7-1
Numbering, 1-15

One or More of, 1-6, 1-15
One-Character Pattern, 1-14
Options
  Diff, **2-1**
Out of memory, **2-7**
Out of space, **1-21**
Output, **2-1**
Output File, 1-16
Output Format, **2-3**

P, 3-1, 3-2, 3-6, 3-**26**, 3-27
Pass to MS-DOS, **3-24**
pat.h, 1-17, 1-18
PATTERN, 1-17, 1-18, 3-9, **3-12**, 4-1
Pattern ill-formed: no terminating double quote, **1-21**
Period, 1-2, 1-4, 1-13, 1-14, 3-3, 3-5, 3-8, **3-10**,
        3-24, 3-25, 3-27, 6-3, 7-1

Plus Sign, 1-2, 1-6, 1-8, 1-15, **3-10**, 6-3
Prefix, 5-1
Previous Line, 3-6
Previously Matched Pattern, **3-10**
Print Command, 3-2, 3-11
Print Count, 1-15
Print Error Message Command, **3-26**
Print Line Number Toggle Command, **3-26**
Print Names, 1-15
Print No Match, 1-15
Print Number of Current Line, 3-6
Print Range of Lines Command, **3-26**
Print Substitution Results, 3-27
Print the Current Line, 3-6
Print Togle Command, **3-26**
Printable Characters, 2-1, **2-2**
Prompt Toggle Command, **3-26**

q, 3-9, 3-22, **3-26**
Question Mark, **3-10**, 3-12, 3-13, 3-14, 3-26
Quit Command, 3-9, 3-22
Quotation Mark, 1-8, 1-14, 1-20, 1-21, 1-21

r, 1-14, **3-27**
Range of Characters, 1-5
Range of Lines, 3-4
RE, 1-18
Read Text Into Buffer Command, **3-27**
Redirection, 5-1
Redirection of Output, 1-16
Regular Expression, 1-2, 1-18, 4-1, 4-3, 6-3
Regular Expression Match, **1-18**, **1-19**
Regular Expression Pattern, **1-17**
Repeat, 5-2
Replace, 5-2
Replacement String, 3-9
re_gen, **1-17**, 1-18, 1-19
re_match, 1-17, **1-18**, 1-19
re_smatch, 1-18, **1-19**
Right Bracket, 1-2, **3-10**
rm, 6-1
rm -rf * Command, 6-5
Root, 5-1

s, 1-3, **3-27**
Save a File, 3-8
Search Backward, 3-12, 3-14
Search Forward, 3-14
Searching, **3-12**
Semicolon, 3-13
Show File Names, 1-15
Simple Patterns, **1-3**

Single Character Pattern, **3-10**
Single Quotation Mark, 3-26
Single Quote, **3-10**
Size Limitations, **3-23**
Sort, 5-1
Space, 1-8, 2-1
Space Character, 1-3
Spaces, 1-14
Special Characters, **1-2**
Special Symbols, **3-9**
Splat, 1-1, **4-1**
Splat Examples, **4-2**
Split a Line, 3-27
Star, 1-2, 1-6, 1-8, 1-13, 1-15, 6-3
Star Command, 6-5
Statistics, 2-1
Stream Editor, **4-1**
Subdirectory, 6-2
Substitute, 3-8
Substitute Command, **3-27**
Substitution, **3-15**
Substitution String, 4-1
Subtraction Operator, **3-10**
Suffix, 5-1

t, 1-3, 1-14, **3-27**
Tab, 1-3, 1-8, 1-14, 2-1, 4-3
Table Size, **2-1**
Targets File, 3-21
Text Editor, 3-1
The **global** Command, **3-17**
Too few arguments to **Grep**, **1-21**
Too many file names:, **2-7**
Translated Mode, 2-1

Unrecognized option, **2-7**
Uploading, 2-3
Using **Build**, **5-1**
Using **Splat**, **4-1**
Using the **Files** Command, **6-1**
Using the **Grep** Functions in a Program, **1-10**

v, 3-19, 3-25, **3-27**
Verbose, 4-2
Version, 1-15

w, 3-1, 3-8, 3-23, **3-28**
Wc, **2-1**, 7-1
Whitespace, 1-14, 2-1, **2-2**, 2-3, 2-6
Wildcard, 1-15, 1-16, 5-1, 6-3
Wildcard Character, **1-4**
Word Count, **2-1**

WordStar, 7-1
Wrap-around, 3-7
Write Command, 3-2, 3-8
Write Lines to File, **3-28**

x, 1-3, 1-14, 1-21

Zero or More of, 1-6, 1-7, 1-15

[, 1-2, 1-5, 1-13, 1-14
[ ], 1-14, **3-10**

\, 1-3, 1-14

], 1-2, 1-5

^, 1-2, 1-8, 1-13, 1-14, **3-10**, 3-13, 5-2, 5-3, 5-4
^C, 3-2, **3-10**
^G, 1-3

Lattice

Lattice, Incorporated
Post Office Box 3072
Glen Ellyn, Illinois 60138
TWX 910-291-2190
Telex 532253

Lattice is a registered trademark of Lattice, Incorporated.

Printed in USA